Titre: Title:	Sécurité des agents mobiles : protocole sécuritaire à base d'agents sédentaires coopérants
Auteur: Author:	Abdelhamid Ouardani
Date:	2004
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Ouardani, A. (2004). Sécurité des agents mobiles : protocole sécuritaire à base d'agents sédentaires coopérants [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/7506/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/7506/
Directeurs de recherche: Advisors:	
Programme: Program:	Non spécifié

UNIVERSITÉ DE MONTRÉAL

SÉCURITÉ DES AGENTS MOBILES : PROTOCOLE SÉCURITAIRE À BASE D'AGENTS SÉDENTAIRES COOPÉRANTS

ABDELHAMID OUARDANI
DÉPARTEMENT DE GÉNIE INFORMATIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
Juillet 2004

Your file Votre référence
ISBN: 0-612-97972-5
Our file Notre référence
ISBN: 0-612-97972-5

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

SÉCURITÉ DES AGENTS MOBILES : PROTOCOLE SÉCURITAIRE À BASE D'AGENTS SÉDENTAIRES COOPÉRANTS

présenté par : OUARDANI Abdelhamid

en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées

a été dûment accepté par le jury d'examen composé de :

M. ROY Robert, Ph. D., Président

M. PIERRE Samuel, Ph.D., membre et directeur de recherche

Mme. BOUCHENEB Hanifa, Doctorat, membre et codirectrice de recherche

M. FERNANDEZ José, Ph. D., membre

DÉDICACE

À ma famille …

REMERCIEMENTS

Je désire remercier mon directeur de recherche, le professeur Samuel Pierre, et ma codirectrice de recherche, la professeure Hanifa Boucheneb, pour leur soutien, leur patience, leurs conseils et leurs commentaires.

Je désire ensuite remercier tous les membres du Laboratoire de Recherche en Réseautique et Informatique Mobile (LARIM), pour leur support. Et en particulier, je remercie Abdelmorhit El Rhazi pour son aide et ses critiques.

Je désire également remercier ma famille et mes amis pour leurs encouragements et leur soutien.

RÉSUMÉ

Les agents mobiles sont des entités logicielles qui peuvent migrer d'une machine à une autre afin d'exécuter une tâche bien déterminée. Ce nouveau paradigme a révolutionné la manière de procéder dans les applications classiques centralisées ou même celles du client/serveur. En effet, selon cette nouvelle architecture, l'exécution des programmes se fait auprès des données, ce qui a apporté des avantages tel que la réduction de la charge réseau. Aussi, avec le développement spectaculaire de l'Internet, ces entités minuscules ont favorisé l'utilisation des unités mobiles, qui sont certes limitées en terme de capacité, mais très pratiques dans des environnements répartis. De plus, ces entités s'adaptent mieux aux nouveaux besoins en termes de mobilité des usagers.

Cependant, cette architecture est souvent confrontée à des problèmes de sécurité. En effet, les plates-formes peuvent faire l'objet d'attaques de la part d'agents malveillants. Aussi, des plates-formes malicieuses peuvent menacer la sécurité des agents qu'elles accueillent. Des approches de sécurisation des agents ou des plates-formes ont vu ainsi le jour. Par ailleurs, de nouveaux types d'attaques continuent à apparaître en exploitant d'autres vulnérabilités de l'architecture et mettant ainsi les approches développées en échec.

C'est dans ce cadre que s'inscrivent les travaux effectués dans ce mémoire, avec pour objectif de concevoir un protocole qui permet de sécuriser l'agent mobile. Ensuite, il s'agit de le valider formellement, l'implémenter et en évaluer les performances. Pour ce faire, et dans un premier temps, nous avons examiné les attaques qui menacent la sécurité de cette architecture. Ainsi, nous avons relevé les spécificités de chaque attaque. Dans un deuxième temps, nous avons étudié les approches de protection. À la base de cette étude, nous avons conçu notre protocole pour combler les limitations d'un

protocole de base. Nous avons ainsi élaboré un protocole qui combine entre quatre concepts : la coopération entre les agents mobile et sédentaire, le comportement de référence (plates-formes de confiance qui abritent nos agents sédentaires coopérants), la cryptographie et la signature numérique pour sécuriser les communications inter agents et l'exécution limitée dans le temps (compteur de temps). Précisément, nous avons mis en œuvre une approche d'estimation dynamique d'un compteur de temps pour la détection de l'attaque de ré-exécution de code. Aussi, nous avons pris en charge l'attaque de modification permanente de code en terme de détection et de protection. Par ailleurs, nous avons doté le protocole de mécanisme de survivabilité qui le rend ainsi robuste et tolérant aux fautes. En effet, d'un côté, nous avons utilisé un agent sédentaire de reprise qui permet de prendre la relève en cas de panne du premier agent sédentaire coopérant. D'un autre côté, en cas de déni de service de l'agent mobile, ce dernier est tout de suite remplacé par un clone qui prendra la relève là où l'autre est éliminé. Nous avons ensuite analysé la sécurité de notre protocole en passant en revue l'ensemble des attaques possibles.

Après cette étape, nous avons vérifié formellement notre protocole en utilisant la technique du model-checking implémentée par l'outil UPPAAL. Tout d'abord, nous avons représenté le protocole sous forme de réseau d'automates temporisés. Ensuite, nous avons conçu un processus qui modélise une plate-forme qui attaque l'agent mobile sans avoir à changer ni sa structure ni son comportement. Finalement, nous avons spécifié les propriétés formelles que le protocole devrait satisfaire. Ces propriétés relèvent soit de la survivabilité/disponibilité, soit de la vivacité ou de l'intégrité du protocole. Nous les avons vérifiées par le model-checker utilisé.

Pour tester notre protocole, nous l'avons alors implémenté dans la plate-forme d'agents mobiles Grasshopper. Nous avons démontré la qualité de notre approche d'estimation dynamique du *Timeout* dans deux cas de figure (cas sans cache - sans proxy, et cas sans cache - avec proxy). En effet, nous nous sommes comparés au protocole de base dans le cas d'une plate-forme attaquante par une ré-exécution de code, et les résultats ont été satisfaisants. Aussi, le comportement de notre protocole en

présence d'attaque a été conforme à nos attentes. Finalement, nous avons mesuré le trafic généré par notre protocole. Nous avons constaté qu'il y a une augmentation de 13% par rapport au trafic total généré. Cela est dû à l'utilisation de l'agent sédentaire de reprise. Cependant, cette augmentation constitue le prix à payer pour avoir un protocole plus sécuritaire et plus robuste.

ABSTRACT

Mobile agents are program instances which can autonomously migrate from one agent platform to another, thus performing tasks on behalf its owner. This new paradigm let agent to run near data what brought advantages such as the reduction of network load. Moreover, with a spectacular development of the Internet, this architecture promotes the use of the mobile units, who are, admittedly limited in terms of capacity, but are very practical in distributed environments. Despite its many benefits, mobile agent technology results in significant security threats from malicious agents and hosts.

In this thesis, we aim to conceive a secure protocol which protects mobile agent against malicious hosts. Then, we will formally validate, implement and evaluate the performances of our protocol. First of all, we investigate threats and countermeasures. Then, based on this study, we have, conceived our protocol to fill the limitations of a basic protocol. We thus designed a protocol which combines four concepts: the co-operation between mobile agent and sedentary agent; the reference execution (reliable platforms which shelters our co-operating sedentary agents); the cryptography and the digital signature to make safe the inter agent's communications and the time-limited execution (*Timeout*). We designed a dynamic approach which estimates a timer to make it possible to detect a mobile agent's code re-execution. Also, we dealt with the attack of agent permanent modification. Moreover, we make the protocol survivable so it will be robust and fault tolerant. Indeed, on one side, we used a recovery sedentary agent which guarantees the handoff in case of breakdown of the primary co-operating sedentary agent. On another side, in case of a denial of service attack, the mobile agent is immediately replaced by a clone which will continue with the same state. We then analyzed the security of our protocol while reviewing the whole of the possible attacks.

After this stage, we formally verified our protocol by using the model-checking technique implemented by UPPAAL. First of all, we represented the protocol by a

network of timed automata. Then, we conceived a process which generates attacks against the mobile agent without any need to change neither its structure nor its behavior. This process can also generate failure on reliable platform. Finally, using CTL, we specified formal properties that our protocol should satisfy. These properties concern survivability and availability, promptness and integrity of the protocol. We then checked them by the model-checker of UPPAAL, and all are satisfied.

To test our protocol, we implemented it in Grasshopper mobile agent's platform. Then, in two scenarios (without cache and without proxy; without cache and with proxy), we assess our approach of dynamic estimation of *Timeout*. Indeed, we compared our results to the basic protocol in case of malicious host who re-executes the mobile agent's code, and the results were very satisfactory. Also, the behavior of our protocol with all possible attacks was compliant to our expectation. Finally, we measured the traffic generated by our protocol. We noted an increase of 13% compared to the total traffic generated. That is due to the use of the recovery sedentary agent. However, this increase represents the cost of having a secure, reliable and more powerful protocol.

TABLES DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

CHAPITRE I

INTRODUCTION

Avec l'explosion qu'a connue l'Internet en termes d'utilisation et de déploiement, le besoin de développer des applications pouvant être exploitées sur une telle architecture fondamentalement hétérogène et très disparate, a connu un énorme intérêt. L'architecture client/serveur a bien démontré son habilité et sa performance à accompagner ce développement. Toutefois, cette architecture s'est trouvée en face des problèmes d'évolutivité et de maintenabilité et a présenté ainsi des limitations quant à l'encombrement des réseaux par des messages requête/réponse. Ces limitations se sont aggravées surtout avec les terminaux et périphériques portables qui, en plus, ont ajouté d'autres exigences en termes de limitation du temps d'exécution et de connexion, des ressources mémoires et de largeur de bande. Ainsi est venu le paradigme « agent mobile » qui se veut une alternative efficace et mieux adaptée aux nouvelles exigences. Pour atteindre un degré de maturité et de complétude, il faut le doter de mécanismes de sécurité robustes qui feront face à ses vulnérabilités. Cependant, ces applications sont beaucoup plus ambitieuses (commerce électronique mobile) au point qu'il faut les sécuriser davantage et combler les failles que connaissent les systèmes d'agents mobiles sur le plan de la sécurité. C'est dans ce contexte que s'inscrit ce mémoire. Il se propose de répondre et de contribuer à la résolution des défis posés par la sécurité des agents mobiles. Dans ce chapitre, nous allons introduire tout d'abord les concepts de base de ce nouveau paradigme. Puis, nous présentons les éléments de la problématique. Ensuite, nous enchaînons par une définition des objectifs de recherche et nous esquissons notre méthodologie de recherche. Pour finir, nous présentons le plan de notre mémoire.

1.1 Définitions et concepts de base

Une architecture d'agents mobiles est composée principalement de deux acteurs : les agents mobiles et la plate-forme. Un agent est une entité logicielle qui peut agir en faveur d'une autre entité (une personne, un autre agent). Il est destiné à accomplir certaines tâches. Il réagit face aux événements externes provenant de son environnement. Ainsi, on peut lui associer plusieurs aptitudes à savoir : l'autonomie, la mobilité, l'aptitude sociale et la coopération, la réactivité, la continuité temporelle, l'adaptabilité, le risque et la confiance. Un agent mobile est un agent capable de migrer entre les nœuds du réseau selon un *itinéraire* qui est soit statique (i.e. préalablement établi) ou dynamique. L'agent est composé de trois parties : le *code* qui est le programme qui l'implémente, *l'état* d'exécution du programme et *les données*. On distingue deux types de migration : *sens faible* quand seulement le code de l'agent migre à sa destination, *sens fort* quand l'agent mobile effectue ses migrations entre les différents nœuds tout en transportant dynamiquement avec lui ses données et son état en plus de son code. Un agent mobile fait intervenir deux entités : son *auteur* (créateur) et son *propriétaire* (utilisateur détenteur de l'agent).

Une plate-forme est l'environnement d'exécution qui est une application tournant au dessus du système d'exploitation. Pour la plupart d'entre elles, elles consistent en une collection de librairies Java (exemple : Grasshopper, Aglet, Concordia, Voyager). Sinon ce sont des applications écrites en langage de script avec interpréteur et des utilitaires d'exécution (exemple : D'Agent, Ara). La plate-forme permet de créer des agents mobiles, leur offre les éléments nécessaires à leur exécution, les accueillie et leur permet de migrer vers d'autres plates-formes.

1.2 Éléments de la problématique

Un aspect de problématique qui survient est celui de la sécurité: la sécurité des agents mobiles et celle des plates-formes sur lesquelles ils s'exécutent. Ce problème ne

cesse d'entraver l'épanouissement et le grand déploiement de telles architectures pour qu'elles couvrent de plus en plus de domaines d'application.

La sécurité des plates-formes est souvent menacée par ces agents itinérants qui se déplacent sans cesse. Ces menaces ont tiré l'alarme et attiré l'attention des chercheurs sur des attaques, évidemment connues depuis bien longtemps dans les systèmes militaires classiques (cheval de Troie), mais sans pouvoir outrepasser et traverser les pares-feux des systèmes informatiques du fait qu'il s'agissait de programmes statiques. Les premiers problèmes, se sont manifestés avec l'apparition du code mobile (Applets et servlets), puisqu'ils arrivent de l'extérieur avec de mauvaises intentions; encore ici, ces menaces restaient limitées et facilement contrôlables. Avec les agents mobiles, la menace à la sécurité des plates-formes ([JAN99], [JAN00]) a pris de l'ampleur.

De nombreuses recherches ([FAR96], [ORD96], [GRE98], [NEC98], [JAN00]) se sont concentrées sur la question pour essayer de cerner cette problématique et ont réussi en grande partie à éliminer et limiter les attaques potentielles perpétrées par des agents mobiles malicieux contre les plates-formes.

L'autre facette de la sécurité de cette architecture concerne les agents mobiles. Contrairement à la protection des plates-formes qui converge déjà vers une maturité vu qu'elle est inspirée dans beaucoup de ses approches de sécurité de celles adoptées dans les systèmes conventionnels, on constate que la protection des agents mobiles est difficile face à des plates-formes malicieuses. Le problème prend de l'ampleur du fait qu'étant donné qu'elles sont les environnements d'exécution des agents mobiles, les plates-formes détiennent tous les éléments nécessaires à leur bon fonctionnement. De plus, elles peuvent accéder aux différents composants des agents mobiles à savoir le code, les données et l'état. Ainsi, elles peuvent les manipuler, les espionner ou commettre toute autre sorte d'attaques, et de là, elles porteront atteinte à leur intégrité, authentification, responsabilité et réputation qui représentent en fait les critères et les exigences devant être garanties dans toute politique de sécurité [JAN00][BIE02].

Certains chercheurs ont traité l'aspect sécurité des agents mobiles en étudiant les attaques [JAN99][JAN00][BIE02]. Ainsi, ils ont essayé de déterminer les différentes

classes d'attaques possibles de la part d'une plate-forme malicieuse, et de trouver les différents aspects qui relient les attaques entre elles [SAN98][ELR02] et de connaître leur importance [ELR02]. Les attaques d'espionnage, par exemple, sont les plus dangereuses car elles peuvent survenir en premier et ainsi ils peuvent mener à d'autres attaques, comme par exemple les attaques de modifications. Et c'est évident, car avant de se proposer pour suggérer des solutions à un problème, il faut d'abord le cerner et le comprendre, et c'est ainsi que des chercheurs ont essayé de proposer des approches pour protéger les agents mobiles, mais malheureusement sans pour autant trouver une mesure de protection si complète et efficace.

Chaque méthode vise un ou plusieurs éléments de l'agent pour assurer leur protection. Les méthodes existantes diffèrent par leur capacité soit à protéger l'agent et à l'immuniser contre les attaques potentielles considérées, ou tout simplement à les détecter. À ce stade, il est intéressant de préciser que l'instant de la détection est soit immédiatement après que l'attaque a été commise ou quelque part, à des étapes ultérieures de la mission de l'agent mobile (par exemple lors d'un autre saut ou complètement après son retour à sa plate-forme d'origine). Il faut noter aussi que même si la protection est incomplète, elle limite les dégâts et adoucit ainsi les conséquences. Certains chercheurs proposent des approches basées sur le calcul par des fonctions cryptographiques [SAN98] qui vise à exécuter l'agent sur de plates-formes sans que ces dernières puissent comprendre le code, le problème est qu'on ne peut trouver des fonctions cryptographiques à tous les problèmes. D'autres proposent des approches qui mènent à protéger l'état ou le comportement de l'agent par rapport à un comportement de référence [FAR96][MIN96][VIG98][HOH00], ces approches permettent de détecter si une attaque est perpétrée ou non et même en cas de collaboration de certaines plates-formes [MIN96], alors que dans le cas de [VIG98] sa proposition stipule que la plate-forme ne doit pas mentir quant aux entrées de l'agent pour que la détection soit possible.

1.3 Objectifs de recherche

L'objectif principal de ce mémoire est de concevoir, valider et implémenter un nouveau protocole sécuritaire qui se veut une continuation du travail *d'Elrhazi* [ELR02]. Ce travail a consisté en la conception, la validation et l'implémentation d'un protocole sécuritaire d'agent mobile basé sur un agent sédentaire parfaitement coopérant dans un contexte de plates-formes malicieuses. Afin d'améliorer ce protocole de base, ce mémoire s'est donné comme objectif de :

- Concevoir des mécanismes performants d'estimation du compteur de temps *Timeout* et l'intervalle d'attente supplémentaire (*IAS*) ;
- Concevoir et spécifier des mécanismes de reprise de l'agent mobile en cas d'attaques de déni de service et de modification permanente de code ;
- Concevoir et spécifier des mécanismes de tolérance aux fautes de la plate-forme de confiance qui abrite l'agent sédentaire ;
- Vérifier formellement le nouveau protocole ;
- Implémenter le nouveau protocole et évaluer son efficacité et ses performances.

1.4 Esquisse méthodologique

Le problème de la tolérance aux fautes intervient à trois niveaux : l'agent mobile, l'agent sédentaire et la plate-forme de confiance. Pour l'agent mobile, une fois capturé ou éliminé, l'agent sédentaire, connaissant tous les résultats à date, pourrait soit cloner l'agent mobile et le doter de tous les éléments qu'avait son prédécesseur pour qu'il poursuive la mission à partir du point où son ancêtre s'est arrêté, soit alerter le propriétaire qui va prendre les mesures nécessaires. Puisque la coopération est déterminante, la mise en service d'un ange gardien pour l'agent sédentaire pourrait bien éviter l'interruption de la coopération avec l'agent mobile.

Concernant les mécanismes d'estimation de temps, une première idée est de construire un modèle qui pourrait prendre en considération différents paramètres entrant

en jeu pour la détermination des compteurs temporels (nombre d'agents actifs sur la plate-forme, charge réseau, latence, ...). Sinon, si la construction d'un tel modèle s'avérerait difficile ou biaisée, le recours à une approche dynamique et en temps réel pourrait donner des résultats aussi satisfaisants.

En vue de valider notre protocole, nous allons en construire tout d'abord un modèle formel, puis nous allons en spécifier les propriétés à l'aide de logique temporelle. Enfin, nous allons le valider par la technique du model-checking.

Pour implémenter notre protocole, nous allons utiliser la plate-forme Grasshopper, ainsi que le langage Java.

1.5 Plan du mémoire

Le chapitre 2 présente la problématique de la sécurité dans les systèmes basés sur la technologie d'agents mobiles. Il expose les différentes menaces à la sécurité ainsi que les principales techniques qui sont suggérées dans la littérature pour faire face aux attaques potentielles. Le chapitre 3 décrit le protocole de base. Il présente aussi la description de nos améliorations concernant les mécanismes de tolérance aux pannes et ceux de l'estimation des compteurs temporels; il présente aussi le comportement de notre protocole quant à la protection de l'agent mobile contre les différentes attaques, en particulier le déni de service et la modification permanente de code. Au chapitre 4, nous présentons le modèle formel de notre protocole, son implémentation ainsi qu'une analyse des résultats obtenus. En conclusion, le chapitre 5 fera un résumé des principaux résultats que nous avons obtenus, présentera les limitations de l'implémentation du protocole et finalement proposera des indications pour des recherches futures.

CHAPITRE II

SECURITE DES AGENTS MOBILES

L'avènement de la technologie des agents mobiles a marqué l'informatique distribuée. Elle s'est montrée en tant qu'alternative de taille pour la technologie client/serveur qui a bien longtemps révolutionné l'informatique classique centralisée. Cette nouvelle technologie a apporté tant de commodités quant aux applications distribuées sur le plan mobilité, autonomie ou éventuelle coopération entre agents, aspects ne représentant que quelques unes parmi plusieurs des caractéristiques dont sont dotées ces entités. Le revers de la médaille est celui de l'aspect sécurité : l'absence d'un cadre de travail sécuritaire qui garantit une protection des agents mobiles et des plateformes, entrave et limite les possibilités que peut offrir la technologie des agents mobiles. Dans ce chapitre, nous allons présenter les problèmes de sécurité reliés au concept, les solutions et les protocoles de protections proposés dans la littérature ainsi que leurs avantages et limitations.

2.1 Problème de sécurité dans les systèmes d'agents mobiles

Une architecture d'agents mobiles fait intervenir d'une part, des agents mobiles, entités logicielles dotées de certaines caractéristiques, et d'autre part une plate-forme, environnement doté d'éléments nécessaires pour l'exécution de ces agents. De ce fait, l'étude de la sécurité des systèmes d'agents mobiles peut se faire en considérant les deux volets suivants :

- La sécurité des plates-formes ;
- La sécurité des agents mobiles.

Dans ce qui suit, nous allons faire un survol des différents aspects touchant à la sécurité des plates-formes. Ensuite, nous allons aborder le deuxième volet concernant les

agents mobiles, volet présentant encore des lacunes et suscite ainsi beaucoup d'attention de la part des chercheurs.

2.2 La sécurité des plates-formes

Les principaux acteurs d'une architecture d'agents mobiles sont les agents et la plate-forme sur laquelle ils s'exécutent. Une analyse pertinente de la sécurité consiste à distinguer les menaces potentielles que peut rencontrer la plate-forme, ainsi que les mesures de protections ou de détection en vue d'une prévention.

2.2.1 Les attaques contre une plate-forme

Ce sont les menaces selon lesquelles des entités exploitent les faiblesses des mesures de sécurité des plates-formes pour entreprendre des attaques. Lesdites entités peuvent être des agents mobiles ou d'autres plates-formes. Nous distinguons les trois formes suivantes [JAN99, JAN00]:

1. La mascarade ;
2. Le déni de service ;
3. L'accès non autorisé ;

Dans [ALL01], l'auteur cite d'autres attaques qui entrent dans cette même catégorie, à savoir:

4. Les dégâts ;
5. Le harcèlement ;
6. La bombe logique.

- *La mascarade* : Cette attaque survient quand un agent non autorisé prétend être un autre agent pour accéder à des services ou ressources pour lesquelles il n'est pas autorisé. L'agent malicieux tend par cette attaque à culpabiliser un autre agent en utilisant son identité, ainsi il n'en est pas tenu responsable et ainsi sa réputation n'est pas compromise.

- *Le déni de service* : un agent mobile commet ce type d'attaque en consommant une part excessive des ressources de la plate-forme. Cette attaque est perpétrée soit intentionnellement en exécutant des scripts exploitant les vulnérabilités du système, sinon à travers des erreurs de code. Un agent malicieux peut exécuter un code conçu pour perturber les services offerts par la plate-forme ou dégrader ses performances.
- *L'accès non autorisé* : Les mécanismes de contrôle d'accès sont utilisés pour prévenir que des usagers non autorisés ou des processus puissent accéder à des services ou ressources pour lesquels ils n'ont pas le privilège et les droits d'accès selon la politique de sécurité. Une plate-forme, qui héberge des agents représentant divers utilisateurs et organisations, devrait s'assurer que ces agents n'ont pas les accès en lecture ou écriture pour des données auxquelles ils n'ont pas droit. À titre d'exemple, les données résiduelles stockées au niveau du cache ou dans d'autres endroits de stockage temporaire.
- *Les dégâts* : cette attaque se manifeste lorsqu'un agent détruit les ressources de la plate-forme par modification, reconfiguration ou effacement de la mémoire ou du disque. Ainsi, tous les agents actifs sur cette plate-forme se voient affectés.
- *Le harcèlement* : il survient quand un agent malicieux perturbe les utilisateurs de la plate-forme en leur envoyant ou montrant des propos ou images publicitaires de façon répétitive.
- *La bombe logique* : c'est une attaque qui mène à d'autres attaques comme le harcèlement ou les dégâts. Il s'agit de déclencher l'exécution d'un script quand certaines conditions temporelles et/ou spatiales sont réunies.

2.2.2 Les mesures de protection des plates-formes

Parmi les soucis communs d'une implémentation d'un système d'agents mobiles est le fait d'assurer que les agents n'interfèrent pas entre eux ou avec la plate-forme. Dans [JAN99], les auteurs introduisent le concept de moniteur de référence. Ils s'appuient sur le principe d'isolation des domaines pour l'agent et la plate-forme. C'est

sur ce concept que se sont basées les premières techniques de sécurisation conventionnelles.

Plusieurs techniques de protection de la plate-forme ont vu le jour. G. Allée [ALL01] a fait le bilan suivant des différentes mesures existantes:

- L'authentification des agents [GRE98] ;
- Le carré de sable ;
- Le contrôle d'accès ;
- La vérification du code ;
- L'estimation de l'état [FAR96] ;
- L'historique des hôtes [ORD96] ;
- Le code avec preuve [NEC98] ;
- Les techniques de limitation [GRE98] ;
- Le journal d'audit.

2.3 La sécurité des agents mobiles

Le deuxième volet pour sécuriser une architecture d'agents mobiles concerne la protection des agents. Pour ce faire, il faut connaître, tout d'abord, les menaces potentielles qui peuvent offenser ces entités. Ensuite, prévoir et entreprendre les mesures nécessaires pour faire face et ainsi garantir une meilleure protection.

2.3.1 Les attaques contre un agent mobile

Ce sont des d'attaques dans lesquelles un agent malveillant ou une plate-forme malicieuse menace la sécurité des agents. C'est ce qu'on appelle respectivement attaques *agent-contre-agent* et *plate-forme-contre-agent* [JAN99][AN00] communément appelé le problème des plates-formes malicieuses [GUA00][SAM02][BOR02].

La classe *agent-contre-agent* comprend des attaques telles que :

1. La mascarade ;
2. Le déni de service ;

3. La répudiation ;
4. L'accès non autorisé.

- *La mascarade* : Les communications inter-agent peuvent avoir lieu directement entre les agents. Elles peuvent aussi exiger la participation de la plate-forme via les services qu'elle offre. Un agent malicieux peut tromper un autre agent en prétendant être un autre dans un but de l'arnaquer. À titre d'exemple, l'agent malicieux se fait passer pour un vendeur de services ou de biens, ainsi il peut acquérir des informations privées sur l'agent victime (carte de crédit, compte bancaire…).
- *Le déni de service* : un agent malicieux peut inonder un autre agent par des messages lui causant ainsi un épuisement au niveau des ressources. Il peut rendre cette attaque encore plus dangereuse en se clonant par exemple et augmenter les chances de provoquer le déni de service dans un bref délai avant sa détection.
- *La répudiation* : Cette attaque survient quand un agent qui a participé à une interaction avec un autre agent, réclamerait le fait que l'interaction n'a jamais eu lieu. Elle peut être accidentelle ou intentionnelle. Ceci peut mener à des situations critiques de disputes. Même un processus d'affaire mal conçu peut mener à de telles situations.
- *L'accès non autorisé* : Si les mécanismes de contrôle n'existent pas ou s'il y a une faiblesse au niveau de la plate-forme, alors un agent malicieux pourrait invoquer les méthodes publiques d'un autre agent. Il peut par exemple l'initialiser et changer son comportement.

Dans [JAN99], les auteurs décrivent les différents aspects requis dans une politique de sécurité. Ces aspects sont les mêmes que ceux connus dans les réseaux d'ordinateurs, à savoir : l'intégrité, la disponibilité, la confidentialité, et la responsabilité et l'anonymat. C'est à la base de ces critères que *Bierman et al.* donnent une classification des attaques par des plates-formes malicieuses. On trouve alors les classes suivantes [BIE02]:

- *Attaques d'intégrité* : inclut toute attaque qui modifie ou interfère avec le code, l'état ou les données de l'agent. Elle implique donc une violation de l'intégrité de l'agent mobile. Le motif peut être intentionnel ou accidentel. Nous citons ici les interférences d'intégrité (erreurs de transmission) et les altérations (la corruption, la modification, la mauvaise interprétation, l'exécution incorrecte du code, de l'état ou des données de l'agent, les altérations des communications de l'agent avec d'autres entités).
- *Attaques par refus de disponibilité* : inclut les attaques selon lesquelles un agent autorisé ne peut pas accéder à des objets ou des ressources auxquels il a droit. Ce sont des attaques d'obstruction. Nous citons dans cette classe : le déni de service par délai ou par refus de migration ou élimination.
- *Attaques de confidentialité* : Quand les composants de l'agent sont accédés de façon illégale par une plate-forme malveillante. Sa confidentialité est ainsi violée. Nous citons dans cette classe : l'espionnage (de l'agent ou de ses communications), le vol et la rétro-ingénierie (reverse engineering).
- *Attaques d'authentification* : survient quand un agent mobile ne peut pas identifier et authentifier la plate-forme qui l'exécute. Cette classe inclut la mascarade et le clonage.

Concernant les attaques d'espionnage, Guang et al. distinguent, dans [GUA00], entre deux types d'espionnage *rapide* si l'environnement de l'agent ne détecte pas si ce dernier a été espionnée, sinon c'est un espionnage *tardif ou lent*. Les attaques de type espionnage, dites aussi de reconnaissance, mènent à d'autres attaques comme le vol et le piratage et les attaques de manipulation (code, données ou état), toutes les deux sont basées sur un espionnage rapide réussi.

Le fait que certaines attaques mènent à d'autres, a permis dans [ELR02] de classifier les attaques par importance. Autrement dit, il y a des attaques qui constituent des attaques en elles même. Nous citons à titre d'exemple la modification de code et le déni de service. Alors que la deuxième catégorie (attaques qui mènent à d'autres attaques) rejoint [GUA00] en stipulant que certaines attaques facilitent la tâche à

d'autres menaces. Ces dernières sont généralement plus dangereuses et ont un effet sur l'agent, à titre d'exemple, nous citons l'espionnage en vue d'une rétro-ingénierie.
Comme synthèse, nous avons pu énumérer dix-huit attaques possibles des plates-formes malicieuses contre les agents mobiles [ELR02][BIE02][GUA00]:

1. Espionnage du code ;
2. Espionnage des données ;
3. Espionnage de l'état ;
4. Espionnage des interactions avec les autres agents ;
5. Vol et piratage [COR99] ;
6. La rétro-ingénierie (reverse engineering) ;
7. Manipulation du code ;
8. Manipulation des données ;
9. Manipulation de flux de contrôle ou état ;
10. Manipulation des interactions avec les autres agents ;
11. Exécution incorrecte du code ;
12. Ré-exécution de l'agent ;
13. Mascarade de la plate-forme ;
14. Retour des résultats erronés des appels systèmes effectués par l'agent ;
15. Clonage ;
16. Déni d'exécution ;
17. Délai d'exécution ;
18. Refus de migration.

- *Espionnage du code :* Le code de l'agent peut être lu par la plate-forme du fait qu'elle est l'environnement sur lequel il s'exécute. Donc, toute compréhension du code peut mener à la connaissance de la stratégie de l'agent, la structure physique des données et de l'état au niveau de la mémoire de la plate-forme.

- *Espionnage des données :* L'attaque par une plate-forme malicieuse visant à lire les données privées d'un agent est difficile à détecter vu que ses effets peuvent ne pas se manifester dans l'immédiat. À titre d'illustration, la connaissance des clés secrètes ou du cash électronique peut induire en une violation du domaine privé dans le premier cas et une perte d'argent dans le deuxième.
- *Espionnage de l'état* : Il est difficile de protéger l'état d'un agent contre une attaque d'espionnage. Avant de procéder, il faut protéger le code, car sinon, en connaissant ce dernier, une plate-forme malicieuse peut toujours déduire plus d'informations sur l'état de l'agent sans lire les données.
- *Vol et piratage du code* : Basé sur un espionnage réussi, une plate-forme malicieuse peut voler le code de l'agent mobile. Un objectif visé pourrait être le fait d'essayer de découvrir la stratégie adoptée par le propriétaire ou l'auteur quant à la résolution de certains problèmes persistants étant donné que l'auteur de l'attaque est un concurrent, ou encore de découvrir la politique d'achat et de magasinage,...
- *Rétro-ingénierie (reverse engineering)* : Cette attaque survient quand une plate-forme avec de mauvaises intentions capture l'agent mobile, analyse son code, son état et ses données pour modifier le profile des informations auxquelles il peut accéder, ou encore en vue d'en construire une copie pour effectuer d'autres attaques.
- *Espionnage des interactions avec les autres agents* : vu que la plate-forme est l'environnement d'exécution de l'agent, elle peut ainsi voir toutes ses interactions avec d'autres agents ou plates-formes distantes. Si ces interactions ne sont pas protégées, la plate-forme courante peut détecter l'intention d'achat de la part de l'agent sans même qu'elle voit l'exécution du code.
- *Altération du code* : Un agent qui arrive dans une plate-forme peut subir une attaque de type altération de code. La plate-forme coupable vise, par cette attitude, à changer le comportement par rapport à celui qui lui est assigné.

- *Altération des données* : Une plate-forme malicieuse peut changer les données de l'agent si elle en connaît l'emplacement physique ainsi que la sémantique qui relie ses différents éléments simples.
- *Altération de l'état* : La plate-forme peut influer le comportement de l'agent en manipulant son état, sans qu'elle ait besoin d'accéder à ses données. Par exemple en changeant juste l'état à la $n^{ème}$ instruction, la plate-forme conduit l'agent ainsi à choisir l'offre qu'elle propose.
- *Manipulation des interactions avec les autres agents* : Cette attaque est en général précédée par une attaque d'espionnage des interactions. Une fois l'attaque précédente effectuée, et si la plate-forme actuelle peut manipuler les interactions de l'agent, elle peut rediriger ses interactions (négociation, ordre d'achat...) vers d'autres entités (agents ou plates-formes) avec lesquelles elle coopère. Elle peut même arrêter ses interactions si elle voit que la conclusion de ses négociations peut influer ses propres intérêts, par exemple si l'agent va épuiser ses ressources financières.
- *Exécution incorrecte du code* : une plate-forme malicieuse peut avoir recours à perpétrer une attaque de ce type en changeant soit la façon selon laquelle elle exécutera le code soit simplement les résultats de l'exécution. Ainsi, il peut se passer des attaques d'altération (manipulation de code ou l'état) pour atteindre les mêmes objectifs.
- *Ré-exécution du code de l'agent* : une plate-forme malicieuse qui intercepte l'agent peut le rejouer en partie autant de fois que ceci l'intéresse. Cependant, cette attaque n'est dangereuse que si les opérations effectuées influence l'intégrité de l'agent [ELR02]. L'exemple de la ré-exécution d'un ordre d'achat fait partie de cette catégorie.
- *Mascarade de la plate-forme* : La plate-forme malicieuse peut commettre cette attaque en trompant l'agent sur sa vraie destination. Donc en interceptant l'agent, elle cachera son identité, pour qu'il ne s'en rende pas compte. Une mascarade peut conduire souvent à d'autres menaces de sécurité telle les attaques de lecture (espionnage).
- *Retour des résultats erronés des appels systèmes effectués par l'agent* : Une telle attaque est perpétrée contre l'agent dans un objectif de masquer les vraies informations

de la plate-forme actuelle. À titre d'exemple, une plate forme malicieuse peut masquer son identité pour tromper l'agent visiteur sur sa plate-forme d'accueil.

- *Clonage* : Chaque agent porte une identité unique lui permettant d'avoir le privilège d'accès à certains services ou ressources. Donc, si une plate-forme malveillante crée un clone de cet agent, ce dernier se trouverait face au problème de l'authentification unique (un autre agent s'est fait passer pour lui).
- *Déni d'exécution* : un agent peut être victime d'un tel type d'attaque par un simple refus d'exécution de la part de la plate-forme malicieuse sur laquelle il devrait être exécuté. Ainsi, la plate-forme par exemple vise à ce que l'agent ne se rende pas compte de l'existence d'une meilleure offre sur une autre plate-forme.
- *Délai d'exécution* : Survient quand une plate-forme malicieuse retarde l'exécution d'un agent ou le fait attendre pour un service donné. Elle ne lui permet l'accès qu'après l'écoulement d'un intervalle de temps.
- *Refus de migration* : Quand une plate-forme malicieuse refuse de transmettre l'agent mobile à sa destination escomptée, ainsi, elle le bloque et met fin à sa mission.

2.3.2 La protection des agents mobiles

Pour réduire la vulnérabilité des agents mobiles face à toutes les menaces possibles de la part des entités malveillantes, des techniques ont vu le jour essayant au mieux de protéger ces agents. Au meilleur des cas, les approches qui existent à ce jour arrivent à une protection des agents mobiles face à certaines des attaques, donc une protection partielle, sinon elles se contentent seulement de détecter les attaques. Au pire des cas les attaques passent inaperçues. Ainsi, il y a toujours des menaces potentielles qui portent atteinte à la sécurité des agents mobiles.

Dans ce qui suit, nous allons essayer de faire une synthèse des approches existantes dans la littérature couvrant la protections des agents mobiles.

Parmi les techniques de protection, nous citons :

1. La protection des agents mobiles par cryptographie (Sander et Tschudin, 1998) ;
2. La sécurité dans le système Ajanta (Karnik et Tripathi, 2000) ;

3. La protection des agents mobiles en utilisant les états de référence (Vigna, 1998; Minsky et al., 1996) ;
4. La boîte noire sécuritaire limitée dans le temps (Hohl, 1998) ;
5. Sécurisation basée sur le modèle de poste de police (Guan et al., 2000) ;
6. Modèle de protection trois tiers (Sameh et al., 2002) ;
7. Enregistrement d'itinéraire par agents coopérants (Roth, 1998; Allée, 2001).

La protection des agents mobiles par cryptographie :

Dans [SAN98], Sander *et al.* proposent une technique qui permet à un code d'exécuter correctement certaines primitives cryptographiques même dans un environnement qui n'est pas de confiance. Le code opère sans besoin d'interaction avec sa plate-forme d'origine. L'objectif est que la plate-forme exécute un programme chiffré sans avoir à le déchiffrer et le comprendre. Le problème se présente comme illustré dans le Tableau 2.1.

Tableau 2.1 Calcul par fonction cryptographique

A	B
A détient un programme qui calcule une fonction f	B possède une donnée x
A ne veut pas que B comprenne f	B peut évaluer f(x) pour A.
A chiffre f => E(f) , crée un programme P qui implémente E(f) :P(E(f)) et l'envoie à B	
	B calcule P(E(f))(x) et l'envoie à A
A déchiffre P(E(f))(x) et obtient f(x)	

Ainsi, un agent mobile (*A*) pourrait évaluer *P(E(f))* sur la donnée x de la plate-forme (*B*) sans pour autant que *B* ne sache quoi que ce soit sur le fonction f. Cependant, il faut identifier des méthodes cryptographiques pour toute fonction f. Sander et al.

proposent une méthode uniquement pour les fonctions polynomiale et rationnelles. Les premiers résultats semblaient être prometteurs, mais il reste de trouver une solution globale qui pourrait s'étendre aux autres classes de fonctions.

La sécurité dans le système Ajanta :

Le système Ajanta[1] conçu par Karnik *et al.*[KAR00] est un système d'agents mobiles qui offre une architecture pour leur protection. Pour ce faire, ils proposent pour chaque agent un certificat d'identité assignée par son auteur. Ce certificat contient le nom de l'agent et les noms de son auteur, son propriétaire ainsi que la base du code de l'agent (un URL du serveur qui donne les classes requises par l'agent).

En effet, vu qu'un agent mobile est exposé à la plate-forme sur laquelle il s'exécute, si cette plate-forme est malicieuse, alors l'état de l'agent sera vulnérable aux modifications. Dans ce système, les auteurs essaient d'offrir des mécanismes qui permettent au moins la détection de telles altérations. Alors, trois mécanismes sont offerts : le premier permet à l'auteur de déclarer une partie de l'état de l'agent comme étant en lecture seule, le deuxième permet à l'agent mobile de créer des journaux en ajout seul et le troisième dit *révélation sélective* permet à la plate-forme d'origine de dédier certaines informations de l'agent à des plates-formes bien déterminées. Autrement dit, seule une partie des données de l'agent sera visible à la plate-forme à laquelle elle est destinée.

- *État en lecture seule :* En général, l'agent mobile contient des objets qui ne doivent pas être modifiés. À titre d'exemple, sa carte d'identité qui représente sa crédibilité et qui ne devrait en aucun cas être modifiée, sauf par son propriétaire. Le système Ajanta fournit un objet conteneur (*ReadOnlyContainer*) qui est en lecture seule. Ce dernier contient un vecteur à plusieurs types d'objets signés. À la création de l'agent, ce vecteur est initialisé avec les valeurs appropriées en lecture seule. La structure de ce

[1] Voir http://www.cs.umm.edu/ajanta

conteneur est illustrée dans la Figure 2.1. La signature est calculée en appliquant une fonction de hachage à ce vecteur, puis on chiffre le résultat avec la clé privée (K^-_A, algorithme *DSA*[1]) de la plate-forme d'origine : *Sign* = K^-_A (*Hash(objs)).* La méthode de vérification du conteneur permet à chaque serveur sur le chemin de l'agent de vérifier si l'état en lecture seule est altéré. Pour se faire, il doit accéder à la clé publique (K^+_A) de la plate-forme d'origine et vérifier l'identité suivante : *hash (objs)* == K^+_A *(sign).*

```
class ReadOnlyContainer {
   Vector objs;
   byte[] sign;      // set to K-A( h(objs) )

   // constructor
   public ReadOnlyContainer (Vector, PrivateKey);

   public boolean verify (PublicKey);
}
```

Figure 2.1 Structure du conteneur à lecture seule d'Ajanta

Si une plate-forme antérieure essaye de modifier les éléments du vecteur alors la signature serait corrompue puisque seule la plate-forme d'origine détient sa clé privée. Cependant, si la plate-forme remplace tout l'objet en lecture seule par un autre conteneur (vecteur et signature correspondante), alors nulle autre plate-forme ne peut détecter cette attaque.

- *Journaux en ajout seul :* Ajanta offre un objet en ajout seul qui sert à protéger les objets qui ne doivent pas être modifiés. Il contient un vecteur d'objets dans lequel l'agent mobile peut y ajouter les données collectées ou les résultats obtenus dans les plates-formes visitées. La structure de ce conteneur est illustrée dans la Figure 2.2. En effet, l'agent dispose ainsi d'un journal dans lequel il ne peut qu'ajouter (*appendOnlyContainer*) les données, la signature de la plate-forme, son identité et un

[1] Algorithme libre pour la signature numérique.

checksum qui sert à la vérification ultérieure. Ce *checksum* est en fait obtenu en chiffrant, avec la clé publique de la plate-forme d'origine, les informations à date i.e. l'ancien *checksum*, l'objet et la signature. Initialement, le *checksum* est obtenu en ne chiffrant (clé d'*ElGamal*[2]) que le contenue d'une variable *nonce* (voir figure 2.2) que seule la plate-forme d'origine connaît la valeur. Au retour la plate-forme d'origine procède en ascendance (de la fin au début) pour vérifier les résultats.

```
class AppendOnlyContainer {
   Vector objs;
   Vector signs;
   Vector signers;
   byte[] checkSum;

   public void checkIn (Object X);
   // checkSum = encrypt (checkSum + sig(X) + server)

   public boolean verify (PrivateKey k, int nonce);
}
```

Figure 2.2 Structure du journal à ajout seul d'Ajanta

Cette approche est cependant très faillible. En effet, si par exemple l'agent revisite une plate-forme malicieuse, alors elle pourrait bien effacer toutes les entrées qui ont été effectuées après sa visite précédente. Aussi, l'approche est vulnérable en cas de coopération de deux plates-formes malicieuses. En effet, la plate-forme visitée en premier par l'agent envoie à la deuxième plate-forme le checksum, ainsi la deuxième peut supprimer toute les entrées effectuées entre les deux et ne garder que les leurs.

- *Révélation sélective* : Ce mécanisme est introduit pour protéger certains éléments qui ne seront accessibles que par des serveurs bien déterminés. Pour cette fin, Ajanta offre un état dédié (*Targeted State*) qui est une classe contenant un vecteur

[2] Algorithme similaire à RSA (exploitant un autre problème mathématique), aussi fiable à longueur de clé équivalente, c'est un algorithme libre.

d'objets. Chaque objet est chiffré par la clé publique de la plate-forme à laquelle il est destiné. La structure de cet objet est illustrée dans la Figure 2.3.

```
class TargetedState {
   Vector objs;       // encrypted with server's pub-key
   Vector servers;
   byte[] sign;       // set to K⁻A(h(objs + servers))

   // constructor
   public TargetedState (Vector, Vector, PrivateKey);

   public boolean verify (PublicKey k);
}
```

Figure 2.3 Structure de l'état ciblé d'Ajanta

Les identités des serveurs correspondants sont aussi incluses dans un autre vecteur, ces deux vecteurs sont ensuite hachés et signés par le propriétaire de l'agent avant la première migration de ce dernier. À l'arrivée chez une plate-forme, cette dernière exécute une méthode de déchiffrement qui va extraire tous les objets qui lui sont destinés. Si une plateforme tente de modifier des objets cibles, la signature de l'agent contenue dans le conteneur n'est plus valide.

La protection des agents mobiles en utilisant les états de référence :

Un état de référence [HOH00] est composé des parties variables de l'agent mobile exécuté sur une plate-forme montrant un comportement de référence. On définit un comportement de référence comme le comportement d'une plate-forme qui agit selon ce qu'elle est supposée faire sans commettre d'attaque sur l'agent en exécution. À l'opposé, une plate-forme qui commet une attaque contre un agent, produit un comportement différent de la référence. Ceci se manifeste au niveau de l'état de l'agent. Donc, on peut mesurer une différence de comportement qui nous permettra de détecter d'éventuelles attaques de la part d'une plate-forme malicieuse.

Il y a deux méthodes qui utilisent cette technique : la réplication du serveur de Minsky et al. [MIN96] et les traces d'exécution de Vigna [VIG98].

Dans l'approche *serveurs répliqués*, Minsky et al. proposent, dans [MIN96], d'utiliser un mécanisme de tolérance aux fautes qui détecte également les attaques de plates-formes malicieuses. Ces chercheurs supposent, pour chaque session d'exécution sur une plate-forme, l'existence d'un ensemble de serveurs indépendants et répliqués (qui offrent les mêmes ressources) mais qui ne collaborent pas pour attaquer une plate-forme. Chaque étape de la session d'exécution est effectuée en parallèle par tous les réplicats. Après l'exécution, un vote est effectué pour déterminer les résultats de l'étape (états résultants). On retient alors l'état avec le plus de voix. Évidemment, mêmes (n/2-1) plates-formes malicieuses peuvent être tolérées. Cette approche peut détecter toutes les attaques qui produisent des états différents. Une éventuelle collaboration de (n/2 -1) plates-formes malicieuses, dans une même étape, peut être détectée.

Dans l'approche *traces d'exécution* proposée par Vigna et al. [VIG98], pour vérifier l'exécution d'un agent, la plate-forme produit une *trace*. L'idée est la suivante : quand une fraude est soupçonnée, l'approche permet au propriétaire de l'agent de vérifier la session d'exécution dans différentes plates-formes. Pour ce faire, chaque plate-forme enregistre une trace similaire à celle illustrée dans le Tableau 2.2. Une trace consiste en des paires (n,s) tel que n représente l'identifiant de l'instruction ou la ligne du code exécutée. Si cette ligne modifie l'état de l'agent en utilisant des informations issues de l'extérieur de l'agent, alors s désignera la liste des variables mises en jeu ainsi que leurs valeurs après l'exécution de cette ligne. Si aucune entrée de l'extérieur de l'agent n'est utilisée, alors s est vide. Après l'exécution, la plate-forme crée deux fichiers représentants la trace et l'état résultant. Ensuite, elle leurs applique des fonctions de hachage, les signe et les envoie à la plate-forme suivante avec le code et l'état de l'agent. L'agent continue à effectuer sa tâche avant de revenir à sa plate-forme d'origine. En cas de soupçons, le propriétaire peut décider d'entreprendre une vérification et demande les traces enregistrées auprès des plates-formes visitées en commençant par la première. Alors, le propriétaire effectue un hachage avec la trace reçue et compare les

résultats avec ceux enregistrés chez la plate-forme suivante. S'ils sont identiques, la plate-forme termine alors sur cette trace. Et pour une vérification globale, le propriétaire ré-exécute l'agent selon l'état initial. En cas de ligne utilisant des entrées de l'extérieur, on en utilise les valeurs enregistrées dans la trace. On peut détecter donc si une plate-forme ment ou non en vérifiant si le hachage résultant sur cette plate-forme est différent ou identique à celui signé par celle-ci, et ainsi de suite.

Cette approche détecte toute attaque qui produit un état différent à condition que la plate-forme ne mente pas quant aux entrées de l'agent. Cependant, la détection n'est pas immédiate et la vérification est lourde.

Tableau 2.2 Trace d'exécution

Code		Trace	
10	read(x)	10	x = 5
11	y = x+z	11	
12	m = y+1	12	
13	k = cryptInput	13	k=2
14	m = m+k	14	

La boîte noire sécuritaire limitée dans le temps :

F. Hohl propose, dans [HOH98], une méthode qui consiste en la génération d'un agent exécutable à partir d'un ensemble donné de spécifications qui ne sont pas vulnérables aux attaques d'espionnage ou altération. Cet agent constitue la boîte noire. Selon cette définition, on ne peut voir que les entrées et les sorties de cette boîte noire.

Les mécanismes de conversion qui génèrent un agent avec les propriétés d'une boite noire utilisent des paramètres de configuration qui permettent la création de différentes boites noires pour la même spécification (voir figure 2.4). D'après cette

définition, à tout moment, ni le code ni les données de spécification de l'agent ne peuvent être lus ou modifiés.

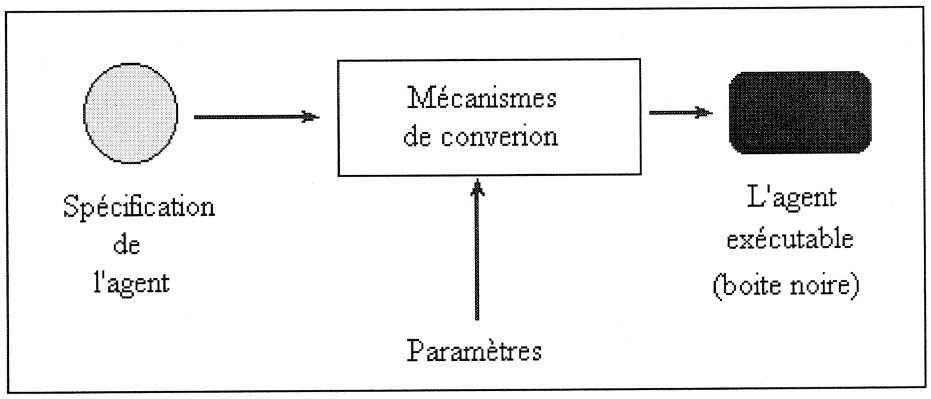

Figure 2.4 Boîte noire limitée dans le temps (F. Hohl)

La boîte noire limitée dans le temps, est définie selon *Hohl*, comme suit :

- Un agent est une boite noire si pour un intervalle de temps donné, le code et les données de spécifications de l'agent ne peuvent pas être lus. Aussi, ils ne peuvent pas être modifiés ;
- Les attaques après l'intervalle de protection sont possibles. Cependant, elles n'ont pas d'effets.

Les approches qui protègent l'agent des attaques des plates-formes malicieuses sont basées sur le fait que celles-ci peuvent lire le contenu de chaque variable ainsi que l'emplacement de chaque ligne de code en mémoire. En général, les attaquants essaient de comprendre la sémantique du code pour en extraire le *modèle mental* [HOH98] de l'auteur. Donc, pour attaquer il faut s'approprier un tel modèle ou le construire avant l'arrivée de l'agent. Une deuxième chose qui est utilisée, ce sont les *algorithmes de confusion* qui rendent le code difficile à comprendre et à analyser. Ces algorithmes devraient prendre en considération les attributs de l'agent qui peuvent être modifiés (les instructions et les données) et les habilités et caractéristiques de l'attaquant qui

dépendent de sa connaissance ou non des spécifications de l'agent. Dans ce cas, il doit analyser la boîte noire pour en construire le modèle mental.

Un des problèmes de cette approche est que l'intervalle de temps que donne l'algorithme de confusion doit avoir une longueur adéquate. Si l'intervalle de protection est long, alors l'agent peut migrer vers plus de plates-formes ou être ré-exécuté abusivement. Sinon, on pénalise alors l'application. Un autre problème relatif à cette approche réside dans la manière adéquate pour déterminer cet intervalle de protection.

Sécurisation basée sur le modèle de poste de police :

Dans [GUA00], les auteurs citent quelques éléments de base qui peuvent être adressés par un model pratique de sécurité pour ainsi résoudre le problème des attaques des plates-formes malicieuses. Ces éléments sont les suivants :

1) les *attaques d'espionnage doivent être prévenues en premier.* Cette assertion est justifiée par le fait que la plupart des attaques sont basées sur un espionnage. Si l'agent est protégé contre l'espionnage, alors la chaîne des attaques qui viennent après ne pourra pas avoir lieu ;

2) *L'espionnage tardif ou lent ne devrait pas être négligé* ;

3) *Les attaques de type élimination doivent être considérées* : le modèle de sécurité doit intégrer un mécanisme pour localiser les agents par l'introduction de serveurs de retracement (aussi utilisés dans [ZHU99]).

En analogie avec le système des postes de police municipaux, les auteurs appellent région un ensemble de plates-formes reliées entre elles par des connections à haut débit et avec les autres plates-formes à l'aide de connections à faible débit. Chaque région comporte une plate-forme spéciale appelée poste de police (*PO*). Cette dernière a pour rôle d'enregistrer tout agent mobile visiteur à la région. La *PO* peut détecter toute élimination ou capture de l'agent et peut prendre des mesures en conséquences. Ils supposent aussi qu'à tout moment, la *PO* est fiable et ne peut attaquer les agents mobiles. Le modèle fonctionne comme suit : avant de migrer vers une plate-forme *H*, l'agent mobile visite tout d'abord la *PO* de la région, s'enregistre et devient actif.

Ensuite, il envoie sa partie esclave (*S*) contenant uniquement des actions non candidates à être attaquées (ex. collecte d'information). Une fois sa tâche terminée, la partie *S* retourne sur *PO* et ainsi l'agent peut effectuer sécuritairement ses actions critiques avec les résultats de *S* avant de choisir sa nouvelle destination. Cependant, la difficulté réside dans la manière de faire le partitionnement des régions. Aussi, la *PO* va certainement subir un encombrement vu que tous les agents vont s'y exécuter pour s'enregistrer et envoyer leurs parties *S* vers leurs destinations. Par ailleurs, les auteurs n'ont pas donné de précisions quant au compteur de temps utilisé et au bout duquel leur modèle considère que l'agent est éliminé.

Modèle de protection trois tiers :

Dans [SAM02], les auteurs ont effectué une combinaison des techniques d'algorithmes de confusion, de chiffrement et d'agent limité dans le temps pour protéger l'agent contre les plates-formes malicieuses. En effet, un agent qui expire, il est soit éliminé, soit réarmé dans le cas où il est retardé suite à des problèmes de réseau. Dans ce cas, il doit vérifier l'identité de la plate-forme de confiance pour lui donner ses jetons qui doivent être remplacés et réinitialisés. L'algorithme de confusion utilisé change le code et les données tout en insérant des lignes de code inutiles dans le code initial rien que pour en rendre la compréhension difficile. Aussi, il modifie les variables numériques en les multipliant par un nombre aléatoirement généré. De la même façon, les valeurs des variables de type chaîne sont altérées en considérant leurs indexes (valeurs numériques). Et comme troisième élément, ils utilisent DES (chiffrement symétrique) comme algorithme de chiffrement des données. Cependant, leur grand problème réside dans le fait que l'agent ainsi généré (avec la sécurité trois tiers) prend beaucoup plus de temps et de ressources mémoire qu'un agent simple, ce qui rend leur protocole moins performant. Aussi, il n'est pas difficile de construire un modèle mental après un tel algorithme de confusion.

Enregistrement d'itinéraire par agents mobiles coopérants :

L'enregistrement d'itinéraire est le fait d'inscrire et vérifier le trajet effectué par l'agent mobile à chaque saut effectué. *Roth* introduit, dans [ROT98], la notion de coopération entre deux agents sans qu'il y ait une plate-forme de confiance pour enregistrer et vérifier l'itinéraire préalablement fixé.

Il définit un ensemble R de plates-formes qui collaborent pour attaquer un agent. Il définit aussi deux ensembles H_1 et H_2 qui ne collaborent pas pour attaquer un agent. Selon cette approche, deux agents 1 et 2 sont dits coopérants si l'itinéraire de l'agent 1 ne contient que des plates-formes de H_1 et que celui de l'agent 2 ne contient que des plates-formes de H_2. Soit h_0 la plate-forme d'origine qui est supposée de confiance, on appelle h_1 et h_2 respectivement les plates-formes actuelles des agents 1 et 2. Si h_1 peut attaquer l'agent 1, alors elle ne peut pas attaquer l'agent 2, et c'est la même chose pour h_2. Il justifie ce choix par le fait que toutes les plates-formes ne sont pas malicieuses et ne collaborent pas pour attaquer un agent. Dans sa spécification du protocole, *Roth* suppose trois hypothèses, à savoir : la première exprime le fait que le transport de l'agent entre deux plates-formes se fait via un canal authentifié. La deuxième stipule que les plates-formes fournissent un canal de communication authentifié aux agents coopérants. Et la dernière permet à l'agent d'avoir un accès, de manière authentifiée, aux identités des plates-formes avec lesquelles il interagit. Le protocole fonctionne selon le schéma suivant :

L'agent 1 enregistre et vérifie l'itinéraire de l'agent 2. Soit $h_i \in H_1$ la $i^{ème}$ plate-forme visitée par l'agent 1 et soit $id(h_i)$ son identité. Soit $Prev_i$ l'identité de la dernière plate-forme où l'agent mobile s'est exécuté et $Suiv_i$ l'identité de la prochaine plate-forme où l'agent compte aller après la plate-forme h_i. L'agent débute et finit (après n sauts) par la plate-forme h_0 ($h_0 = h_n$).

Initialisation : Pour les deux agents, $Next_i$ a pour valeur l'identité de la plate-forme sur laquelle les agents vont s'exécuter. Puis, chacun des agents est envoyé vers sa première plate-forme par l'intermédiaire d'un canal de communication authentifié.

Étape i (i dans {1,...,n}) : L'agent 1 envoie un message à 2 contenant $Next_i$ et $Prev_i$. Ainsi, l'agent 2 apprend $id(h_i)$ et vérifie que $Id(h_i)=Next_{i-1}$ *et* $Prev_i= id(h_{i-1})$. Si c'est le cas, il enregistre $Next_i$ dans l'itinéraire de l'agent.

Ce protocole peut détecter les attaques qui correspondent à l'envoi de l'agent 1 vers une autre destination ($Id(h_{i+1})<> Next_i$), cette attaque est faisable par h_i de deux manières : soit qu'elle peut modifier la communication entre 1 et 2, soit l'envoyer ailleurs (différemment de sa destination réelle). Cependant, quand deux plates-formes successives sur le chemin de l'un des agents collaborent entre elles, on ne peut détecter l'attaque de changement d'itinéraire. De plus, quand une plate-forme reçoit deux agents qui ont la même plate-forme précédente, alors elle peut intervertir leurs agents coopérants en permutant leurs communications.

Dans [ALL01], l'auteur propose une amélioration du protocole d'enregistrement d'itinéraire de *Roth*. Il suggère que l'agent enregistreur 2 se déplace à chaque fois que l'agent mobile 1 se déplace (quand ce dernier contacte l'agent 2 lui annonçant l'identité de la prochaine plate-forme et celle de la plate-forme précédente). Une fois ces informations reçues, l'agent 2 va se déplacer vers une nouvelle plate-forme et attendre la communication de l'agent 1. Il suppose que l'agent 1 possède une liste ordonnée des plates-formes de l'itinéraire de l'agent 2. Il suppose aussi que l'ensemble des plates-formes visitées par l'agent 1 et celui des plates-formes visitées par l'agent 2 sont disjoints. Par ailleurs, le protocole n'offre pas de protection, mais uniquement une détection d'attaque de changement d'itinéraire. Par contre, le problème persiste toujours quand deux plates-formes successives sur le chemin des agents collaborent.

2.4 Éléments de synthèse

D'après cette présentation des différentes approches de sécurisation, le champ se montre très ouvert pour améliorer la protection des agents mobiles dans des

environnements hostiles. Les menaces de sécurité trouvées à ce jour diffèrent selon le danger qu'elles présentent. En effet, certaines se montrent plus dangereuses que d'autres, car une fois commises, nous pouvons nous attendre aux pires conséquences, ce qui va entraver davantage le développement des applications autour de cette technologie. Tous les protocoles et méthodes de sécurisation vues dans cette section ne sont pas parfaits du fait que certains protègent contre certaines menaces, d'autres se contentent juste de détecter quelques unes des attaques, alors, que d'autres attaques passent inaperçues. Une idée pour améliorer la sécurité est de mixer et complémenter certaines méthodes avec d'autres, ainsi on pourrait arriver à des résultats bien meilleurs.

Un autre protocole [ELR02], que nous allons étudier plus en détail dans le chapitre suivant, se montre très attrayant du fait qu'il a su combiner entre la protection de l'itinéraire de l'agent mobile et la protection des résultats intermédiaires. Il se porte garant quant à la protection de l'agent mobile contre des attaques de dénis de service et de ré-exécution. Il combine quatre aspects de sécurisation, à savoir : la coopération d'un agent sédentaire qui s'exécute sur une tierce plate-forme de confiance; le comportement de référence; la sécurisation des communication par chiffrement et signature et l'exécution limitée dans le temps. Néanmoins, certaines améliorations y seront apportées pour le fiabiliser davantage et palier ses limitations. À ce titre, le problème de l'élimination de l'agent mobile, la tolérance aux fautes du protocole et enfin l'estimation dynamique du compteur de temps utilisé sont des aspects qui doivent être repensés.

CHAPITRE III

PROTOCOLE SÉCURITAIRE À BASE D'AGENTS SÉDENTAIRES COOPÉRANTS

Dans ce chapitre, nous nous proposons d'améliorer un des protocoles les plus récents de protection des agents mobiles. Il s'agit bien du protocole de sécurisation basé sur un agent sédentaire parfaitement coopérant [ELR02]. Ce protocole utilise, en plus du concept de coopération entre agents, celui de plate-forme de confiance. Nos préoccupations ici ont trait à la détection de l'attaque de ré-exécution de code, à la robustesse et à la tolérance aux éventuelles pannes. Dans un premier temps, nous décrirons sommairement le protocole original qui sera la base de notre proposition. Ensuite, nous relèverons les défaillances et les limitations du protocole de base, ce qui nous permettra de reformuler ensuite un nouveau protocole prenant en compte les limitations et incluant les solutions proposées. Enfin, nous décrirons son extension par rapport au protocole de base pour la protection des agents mobiles face à d'autres attaques.

3.1 Protocole de sécurisation de base

Le protocole de sécurisation proposé [ELR02] se base sur deux principes intéressants : d'un côté la coopération d'un agent sédentaire, et d'un autre côté le concept de tierce de confiance. Dans cette section, nous présenterons les grands traits du protocole, l'analyse de ses mécanismes de sécurisation ainsi que ses éventuelles limitations.

3.1.1 Hypothèses

Durant tout son parcours, l'agent sédentaire effectue une seule migration vers la plate-forme fiable; son rôle est de veiller à la protection de l'agent mobile lors de ses différents déplacements. Quant à la tierce de confiance, c'est une plate-forme qui garantit une exécution correcte de l'agent sédentaire; ainsi, la sécurité de l'agent sédentaire lui-même n'est pas compromise. Le protocole fait intervenir six éléments :

- la plate-forme d'origine (*O*) : c'est le point de lancement du protocole ;
- l'ensemble (*P*) des plates-formes sur l'itinéraire de l'agent mobile ;
- la plate-forme de confiance (*T*) ;
- l'agent mobile (*AM*) ;
- l'agent sédentaire coopérant (*AS*) ;
- l'agent estimateur (*AE).*

Pour assurer un bon fonctionnement, le protocole de base fait les trois hypothèses suivantes :

- la disponibilité d'une plate-forme fiable ;
- l'établissement des canaux de communication entre l'agent mobile et l'agent sédentaire ;
- l'existence d'un système cryptographique asymétrique.

Il suppose l'existence d'une plate-forme de confiance (*T*) qui est mise à la disposition de toutes les autres plates-formes pour exécuter l'agent sédentaire coopérant (*AS*). Cette plate-forme doit se comporter correctement quant à l'exécution du code de l'agent (*AS*). Elle ne doit pas collaborer avec aucune des plates-formes pour attaquer la plate-forme d'origine (*O*) ou n'importe quelle plate-forme sur l'itinéraire de l'agent mobile. Ces tierces de confiance existent en réalité dans des applications commerciales déployées sur l'Internet et fournissent des services fiables qui peuvent être payants. La deuxième hypothèse est justifiée du fait que, si une plate-forme refuse d'établir un canal de communication fiable pour l'agent mobile vers son agent coopérant, c'est qu'elle est

attaquante. La troisième hypothèse suppose la disponibilité de systèmes cryptographiques asymétriques. *El Rhazi* cite l'exemple de l'infrastructure PGP qui fournit des services cryptographiques à l'aide d'un système asymétrique (clé privée/clé publique). Toutes les plates-formes P_i autorisent l'agent mobile à effectuer le chiffrement des données transmises.

Par ailleurs, le protocole subdivise le code de l'agent mobile en deux parties. Une partie, dite critique, contient le code de l'agent mobile qui peut influencer l'intégrité de l'agent mobile. Une deuxième partie, dite non critique, contient le reste du code de l'agent mobile.

3.1.2 Mécanismes de sécurisation

La plate-forme d'origine (*O*) génère les agents *AM* et *AS*. Elle envoie l'agent sédentaire vers la plate-forme (*T*) dans laquelle il va séjourner jusqu'à la fin du protocole. Elle envoie l'agent mobile, identifié par son identité (*IdAM*), vers la première plate-forme (P_1) sur son itinéraire. Chaque plate-forme (P_i) est identifiée de manière unique à l'aide de son identifiant (Id_i). On note $Prec_i$ la plate-forme qui la précède (d'où provient l'agent mobile avant son arrivée à P_i) et $Suiv_i$ désigne la plate-forme qui la suit (où l'*AM* va migrer à l'occasion de son prochain saut). L'agent mobile exécute son code à l'intérieur de la plate-forme (*Pi*).

À partir de la plate-forme *Pi*, l'agent mobile envoie un certain nombre de messages à l'agent coopérant qui se charge de détecter d'éventuelles attaques contre le code et les données de l'agent mobile. La Figure 3.1 montre les différents éléments du protocole.

Le protocole note $SIG_{Pi}(X)$ la signature électronique d'une donnée *X* par la plate-forme P_i. $E_{Pi}(X)$ et $D_{Pi}(X)$ désignent respectivement l'opération de chiffrement d'une donnée *X* avec la clé publique de la plate-forme P_i et l'opération de déchiffrement d'une donnée *X* avec la clé privée de la plate-forme P_i.

Le protocole fonctionne selon les étapes suivantes :

Initialisation :

La plate-forme d'origine *O* génère l'agent mobile et son agent coopérant. On marque la partie sensible du code comme étant sa partie critique. On la copie dans l'agent sédentaire. Puis, la plate-forme d'origine initialise les variables des deux agents. Elle envoie l'agent mobile à la première plate-forme P_1 de son itinéraire et l'agent coopérant (*AS*) à la plate-forme *T*. Elle exécute un agent estimateur, pour estimer le *Timeout* et l'envoyer à *l'AS*.

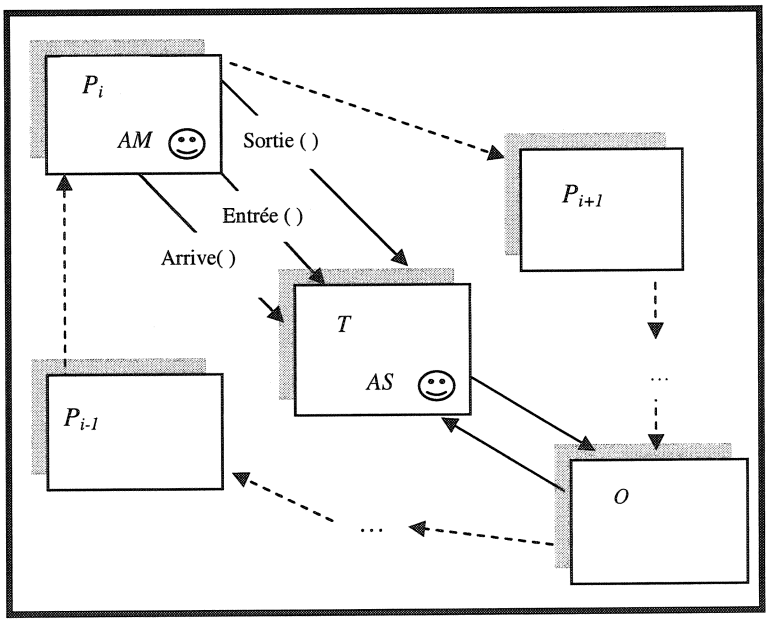

Figure 3.1 Protocole sécuritaire basé sur un agent sédentaire parfaitement coopérant (*El Rhazi*, 2002)

Étape i (i=1, …, L)

Quand l'agent mobile arrive à la plate-forme P_i, il envoie un message *Arrive()* (contenant Id_i , $SIGp_i(IdAM)$, $Prec_i$) à son agent coopérant (*AS*) lui signalant son arrivée à P_i. Alors, l'*AS* vérifie la signature électronique de P_i, puis il vérifie l'itinéraire en vérifiant les égalités suivantes : $Id_i = Suiv_{i-1}$; $Prec_i = Id_{i-1}$. Dans ce cas, l'agent sédentaire initialise un compteur de temps *Timeout* qui va servir à limiter le temps d'exécution de l'agent mobile dans P_i. Cette dernière commence l'exécution du code de l'agent mobile.

Avant d'exécuter la partie critique du code, l'agent mobile envoie un message *Entrée()* chiffré par la clé publique de plate-forme T $E_T(Entrée(Id_i, SIG_{Pi}(X),X))$ à l'agent *AS*. Une fois reçu, l'agent sédentaire le déchiffre avec sa clé privée en calculant $D_T(E_T(Entrée()))$. Il examine ensuite l'identité de la plate-forme P_i en vérifiant $SIG_{Pi}(X)$ sur les données (X). Puis, il exécute la partie dupliquée de l'agent mobile. À la fin, l'agent mobile envoie un message *Sortie()* chiffré $E_T(Sortie(Id_i, SIG_{Pi}(R), R, Suiv_i))$ à l'agent sédentaire. Quand celui-ci reçoit ce message, il le déchiffre avec sa clé privée en calculant $D_T(E_T(Sortie()))$ et obtient le message *Sortie()*. Puis, il vérifie l'identité de la plate-forme émettrice par sa signature $SIG_{Pi}(R)$ sur les résultats. Ensuite, l'agent sédentaire procède à la vérification de l'exécution du code critique de l'agent mobile en comparant les résultats obtenus à l'intérieur de la plate-forme P_i avec ses propres résultats. Si les résultats obtenus sont différents, l'agent sédentaire conclut que la plate-forme P_i a attaqué l'agent mobile. Il marque cette plate-forme comme étant attaquante, enregistre l'itinéraire de l'agent mobile en sauvegardant l'identité $Suiv_i$ de la plate-forme P_{i+1} où l'agent mobile veut migrer.

Étape finale :

L'agent mobile migre vers la plate-forme d'origine O. Celle-ci contacte l'agent sédentaire qui lui envoie la liste des plates-formes attaquantes et les résultats obtenus, puis met fin à son exécution sur T. La plate-forme d'origine O valide les résultats obtenus dans chaque plate-forme visitée. Les résultats des plates-formes marquées comme attaquantes par l'agent sédentaire sont éliminés de la liste des résultats de l'agent mobile. La plate-forme d'origine O met à jour sa liste des plates-formes attaquantes pour qu'elle puisse les éviter lors des prochaines tâches.

3.1.3 Sécurité du protocole

À la réception du message *Arrive()*, l'*AS* arme un temporisateur (*Timeout*) par une valeur appropriée présentant le temps maximum nécessaire à l'exécution de la totalité du code de l'agent mobile et au transfert des deux messages *Entrée()* et *Sortie()* entre l'agent mobile et l'agent sédentaire. Cette valeur est préalablement estimée par

l'agent estimateur (*AE*). Si le *Timeout* expire avant que l'agent *AS* reçoive le message *Sortie()*, l'*AS* déduit que la plate-forme *Pi* est malveillante. L'agent *AS* continue à attendre l'arrivée du message *Sortie()* pendant un autre intervalle de temps supplémentaire (*IAS*). Après l'écoulement de cet intervalle, l'agent *AS* détecte l'attaque déni de service, il avertit la plate-forme *O* pour qu'elle relance le protocole en évitant P_i de l'itinéraire de l'agent mobile. Ensuite, il demande à l'agent mobile de mettre fin à son séjour s'il est capturé. Puis, l'agent sédentaire termine son exécution à l'intérieur de la plate-forme *T*. Si le message *Sortie()* arrive avant l'expiration de l'*IAS*, l'agent *AS* marque la plate-forme P_i comme attaquante et laisse l'agent mobile continuer son itinéraire.

Le problème avec l'estimation du premier temporisateur (*Timeout*) est que, si elle est mauvaise, une plate-forme pourrait être considérée comme malveillante alors qu'elle ne l'est pas. Ceci peut se produire lorsque dans le cas où le *Timeout* est trop court, auquel cas la plate-forme est surchargée vu qu'il y a beaucoup d'agents actifs en même temps, ou encore quand le canal de communication entre les deux agents est relativement congestionné. Dans le cas contraire, une plate-forme malveillante pourrait passer inaperçue alors qu'elle est malicieuse, vu qu'on a accordé plus de temps d'exécution qu'il fallait à l'agent mobile dans cette plate-forme. Ainsi, elle pourrait bien ré-exécuter l'agent mobile sans que l'agent coopérant s'en aperçoive.

Un autre problème qui peut survenir est celui de la tolérance aux fautes. En effet, le protocole ne prévoit pas une reprise à la suite de panne de l'un de ses éléments qui pourrait bien se réinitialiser si l'un des événements suivants survient :

1. un déni de service relatif à l'élimination de l'agent mobile ou de sa capture : le protocole d'*El Rhazi* détecte cette attaque, et il revient à la plate-forme d'origine de relancer le protocole à partir du début, alors que tous les résultats obtenus à date seront perdus. Cela représente une perte au niveau de l'agent mobile et de l'agent sédentaire pouvant même générer des dépenses supplémentaires dans le cas où les services de la plate-forme de confiance sont payants ;
2. une panne de la plate-forme de confiance : dans ce cas, la coopération avec

l'agent sédentaire n'est plus possible. Le mécanisme de protection de l'agent mobile ne sera plus opérationnel, alors qu'il stipulait que l'agent sédentaire doit effectuer les vérifications en vue de détecter d'éventuelles attaques. Ceci pourrait mener à un blocage du protocole. Le protocole d'*El Rhazi* ne prévoit aucune mesure si la tâche attribuée à l'agent sédentaire ne pourrait plus être accomplie suite à une interruption.

Un autre problème se manifeste dans le cas où une plate-forme malveillante perpètre une attaque de type modification de code. L'attaque est immédiatement détectée. Cependant, le problème devient imminent dans le cas où la modification est permanente, ce qui va altérer tous les résultats ultérieurs. Autrement dit, tout le parcours de l'agent mobile devient inutile. Mieux encore, le protocole se trompe sur « l'honnêteté et l'intégrité » des plates-formes ultérieurement visitées. En fait, selon le mécanisme de sécurité utilisé, toutes les plates-formes visitées après cette attaque seront considérées comme attaquantes.

3.2 Description du nouveau protocole proposé

Dans cette partie, nous allons décrire notre proposition. Pour ce faire, tout d'abord, nous introduirons les différents éléments du nouveau protocole. Ensuite, nous exposerons les suppositions sur lesquelles nous nous sommes basées. Nous achèverons cette partie par une description détaillée des différentes étapes du protocole.

3.2.1 Définition des acteurs du protocole

Le nouveau protocole est constitué de trois agents : l'agent mobile, l'agent sédentaire principal et l'agent sédentaire de reprise. Ils seront notés respectivement *AM*, *AS* et *ASR*. Nous désignons par *O*, *T*, *TR* respectivement la plate-forme d'origine, de confiance principale qui abriterait l'*AS*, de reprise qui va accueillir l'*ASR*, et par P_1, P_2,...,P_n les plates-formes constituant l'itinéraire de l'agent mobile. Nous notons $Id(P_i)$ l'identité de la plate-forme P_i, *Temps* le temps d'exécution de l'*AM*, T_{ae} et T_{es} les temps inter-arrivée sur la plate-forme *T* des messages *Arrive()-Entrée()* et *Entrée()-Sortie()*

respectivement. Également, nous désignons respectivement par *Timeout, IAS* et *TimeoutAS* le temps maximum estimé nécessaire à l'exécution de la totalité du code de l'agent mobile, l'intervalle d'attente supplémentaire utilisé précédemment et le temps maximum estimé nécessaire à l'*AS* pour s'exécuter afin de coopérer avec l'*AM* et mettre à jour l'*ASR*.

Nous notons par $E_{Pi}(X)$ l'opération qui consiste à chiffrer une donnée X avec la clé publique de la plate-forme P_i. $D_{Pi}(X)$ désigne l'opération de déchiffrement d'une donnée X avec la clé privée de la plate-forme P_i. Enfin, nous désignons par $SIG_{Pi}(X)$ la signature électronique d'une donnée X par la plate-forme P_i.

La Figure 3.2 montre les différents acteurs du protocole ainsi que les différents échanges entre eux.

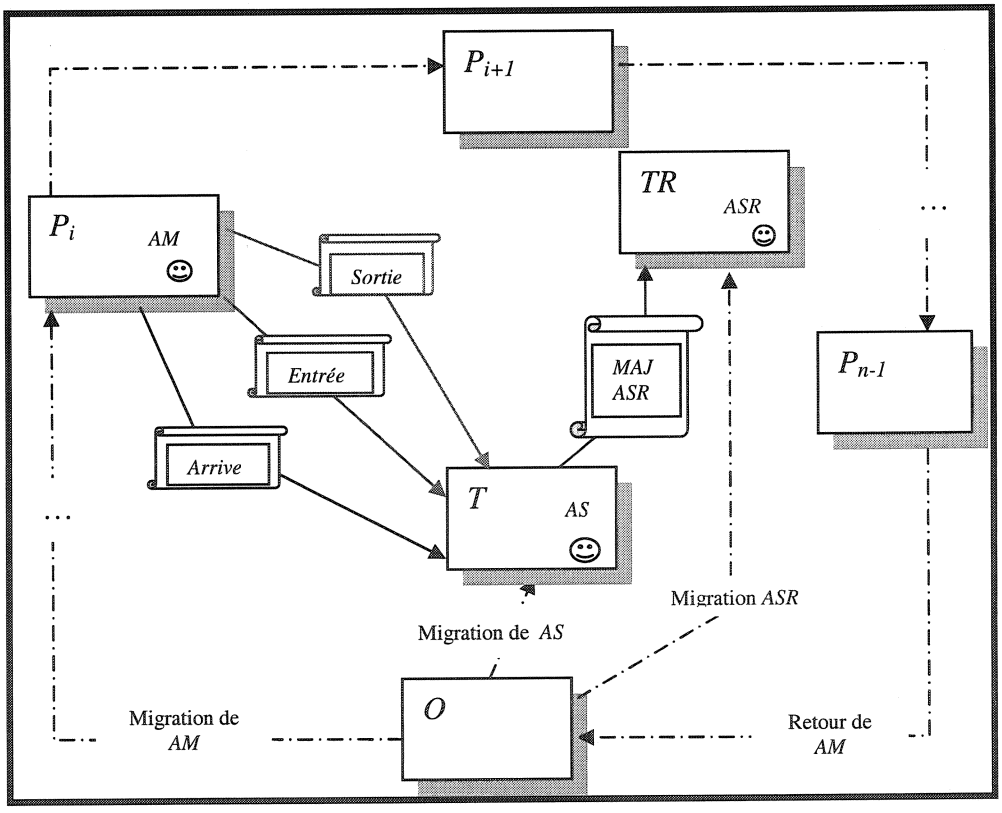

Figure 3.2 Protocole de sécurisation à base d'agents sédentaires coopérants

3.2.2 Suppositions

En plus des hypothèses inhérentes au protocole de base - la disponibilité d'une plate-forme fiable, l'établissement des canaux de communication entre l'agent mobile et l'agent sédentaire et l'existence d'un système cryptographique asymétrique - nous supposons l'existence d'une deuxième plate-forme de confiance pour servir d'accueil à l'agent sédentaire de reprise *ASR* dont nous expliquerons le rôle ultérieurement. Cette hypothèse est similaire à la première hypothèse du protocole de base.

3.2.3 Étape initiale

Le protocole est lancé à partir de la plate-forme d'origine qui crée l'agent sédentaire de reprise *ASR,* l'agent sédentaire *AS* et l'agent mobile *AM.* Elle initialise les variables d'enregistrement d'itinéraire, à savoir $Prec_0$ à $Id(O)$ et $Next_0$ à $Id(P_1)$ dans les agents *AM* et *AS*. Elle initialise les différents paramètres de temps *Timeout*, *IAS* et *TimeoutAS* à des valeurs choisies au préalable. Elle charge la liste des plates-formes attaquantes (de la base de données du protocole) au niveau de l'agent mobile pour qu'il les évite lors de ses sauts. Au niveau de l'agent sédentaire, elle charge une liste de plates-formes alternatives dont nous expliquerons l'utilité plus tard. Enfin, elle fait migrer respectivement *ASR* et *AS* vers les plates-formes *TR* et *T*. Ensuite, elle envoie l'agent *AM* vers la plate-forme P_1. Nous notons à ce niveau que l'*AM* connaît les identités des plates-formes *T*, *TR* et celles de *AS* et *ASR*.

3.2.4 Étape intermédiaire *i*

À l'étape *i,* les deux agents coopérants échangent trois types de messages :

- *Arrive()* : contient l'identité (Id_i) de la plate-forme P_i courante, l'identité de l'agent mobile $SIG_{Pi}(Id_{AM})$ signée par la plate-forme P_i, l'identité ($Prec_i$) de la plate-forme P_{i-1} précédente dans l'itinéraire de l'agent mobile et l'état courant (*EtatAM*) de l'agent mobile;

- *Entrée()* : contient Id_i l'identité de la plate-forme P_i, X les données fournies par P_i comme entrées au code critique et $SIG_{Pi}(X)$ la signature sur les données X en utilisant la clé publique de la plate-forme P_i;
- *Sortie()* : Ce message contient Id_i l'identité de la plate-forme P_i, les résultats R obtenus par l'agent mobile, la signature $SIG_{Pi}(R)$ sur les données R en utilisant la clé publique de la plate-forme P_i, l'identité $Suiv_i$ de la plate-forme suivante P_{i+1} et enfin l'état courant (*EtatAM*) de l'agent mobile.

La Figure 3.3 illustre la structure des trois messages échangés à chaque étape *i*.

À son arrivée à la plate-forme P_i, l'*AM* envoie un message de type *Arrive()* à l'agent sédentaire qui sera en attente de la communication de l'agent mobile. Aussitôt qu'il reçoit le message, l'agent sédentaire vérifie son authenticité en examinant Id_{AM}

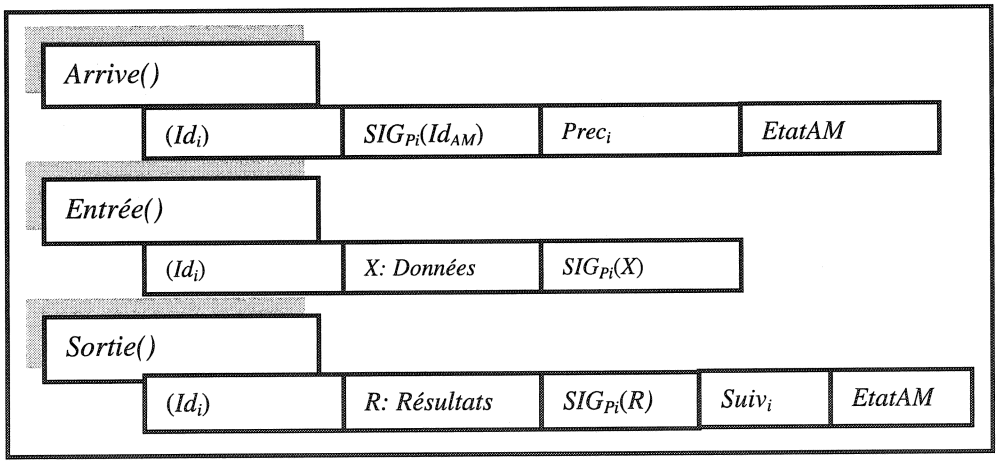

Figure 3.3 Structure des trois messages échangés

signée par la plate-forme P_i. Il vérifie si $\boldsymbol{Suiv_{i-1} = Id_i}$, c'est à dire qu'il s'assure que l'agent mobile a bien migré vers la plate-forme P_i là où il voulait migrer. Sinon, il détecte qu'il y a eu modification d'itinéraire de l'agent mobile. Ensuite, il vérifie l'égalité $\boldsymbol{Prec_i = Id_{i-1}}$ qui l'assure que l'agent mobile provient bien de la plate-forme P_{i-1} d'où il devrait provenir réellement. Dans le cas où les égalités précédentes sont vérifiées,

alors l'agent sédentaire arme le temporisateur (*Timeout*) et attend la réception d'autres messages en provenance de l'agent mobile. Ce temporisateur sert à mesurer le temps d'exécution de l'agent mobile sur la plate-forme P_i; en cas de dépassement de ce temps, l'agent sédentaire sera en mesure de détecter une ré-exécution de l'agent mobile. L'agent sédentaire initialise aussi un compteur de temps qui va mesurer le temps T_{ae} (inter-arrivée des messages *Arrive()* et *Entrée()* en provenance de l'agent mobile).

Juste avant de commencer l'exécution de code critique, l'agent mobile envoie à son agent coopérant un message de type *Entrée()*. Ce message est chiffré avec la clé publique de la plate-forme de confiance *T*. Dans ce message, l'agent mobile fournit à l'agent sédentaire, toutes les informations et les données requises par ce dernier pour l'exécution de la partie critique du code. La description de ce message est donnée à la Figure 3.3. L'agent sédentaire extrait les données *X* signées par la plate-forme P_i avec sa clé privée. Ensuite, il stoppe le compteur de temps et évalue T_{ae}. À l'aide de cette valeur de T_{ae}, l'agent sédentaire calcule à l'avance le temps T_{es} (temps d'inter-arrivée entre les deux messages *Entrée()* et *Sortie())* restant avant l'arrivée du message *Sortie()*. En fait, l'*AS* utilise une fonction *f* implémentant une loi empirique qui déduit T_{es} à partir de T_{ae} ($\boldsymbol{T_{es} = f\,(T_{ae})}$). Ensuite, l'*AS* déduit le temps d'exécution total de l'agent mobile dans la plate-forme courante P_i avec la relation :

$$\boldsymbol{Temps = g\,(T_{ae} + T_{es}) = g\,(T_{ae} + f\,(T_{ae}))}$$

où *g* est une fonction implémentant une loi reliant le temps d'exécution de l'*AM* à la somme des inter-arrivées des messages *Arrive()-Entrée()* et *Entrée()-Sortie()*. Cette valeur *Temps* est ensuite corrigée par un coefficient multiplicateur de sécurité déterminé lors d'expériences du protocole tout comme *f* et *g*, pour enfin obtenir la valeur du nouveau *Timeout* qui prend en considération les caractéristique de la plate-forme P_i en termes de performance, de nombre d'agents et aussi la charge des canaux de communication entre P_i et *T*. De là, l'*AS* réajuste la valeur du *Timeout* déjà initialisée à une valeur préalablement estimée pour détecter une éventuelle attaque de ré-exécution

de code. En fait, l'agent sédentaire détermine le temps au bout duquel il devait s'attendre à recevoir le message *Sortie()* et au delà duquel une attaque est détectée (cf. Figure 3.4).

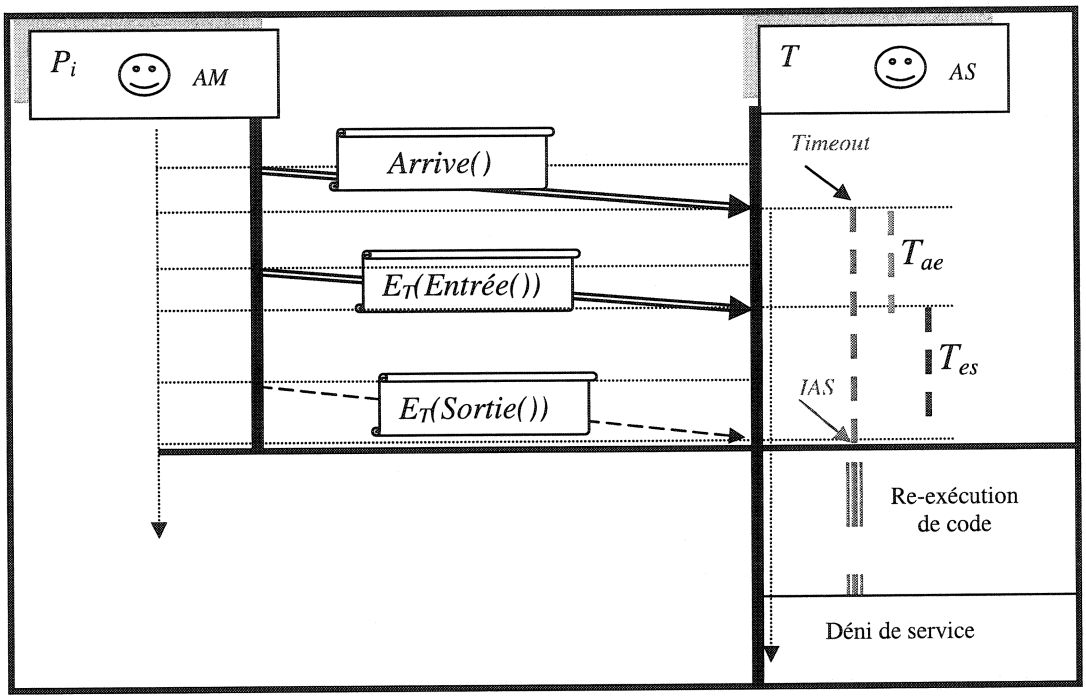

Figure 3.4 Estimation du *Timeout* à partir de l'inter-arrivée *Arrive()-Entrée()*

Alors que l'agent mobile continue son exécution, l'agent sédentaire commence lui aussi à exécuter la copie du code critique de l'agent mobile, étant donné qu'il a reçu toutes les entrées nécessaires. Une fois terminé, l'agent mobile envoie à son agent coopérant un message de type *Sortie()* ($Id(P_i)$, R, $SIG_{Pi}(R)$, $Suiv_i$, $EtatAM$). Il le chiffre avec la clé publique de la plate-forme T en y incluant l'identité de P_i, les résultats obtenus R, une signature sur ces résultats (avec la clé privée de P_i), l'identité de la prochaine plate-forme P_{i+1} où il compte migrer et enfin son état courant (voir Figures 3.2 et 3.3). À la réception de ce message, l'agent sédentaire le déchiffre en utilisant la clé privée de la plate-forme T et vérifie son authenticité. Puis, il extrait les résultats et vérifie la signature de la plate-forme P_i. Ensuite, il compare les résultats reçus avec ceux

qu'il a obtenus. S'ils ne sont pas identiques, la plate-forme est alors considérée comme attaquante de modification de code. Enfin, l'*AS* enregistre ces informations (résultats intermédiaires, itinéraire, crédibilité de P_i) et attend que l'agent mobile migre vers la prochaine plate-forme.

Comme nous l'avons mentionné précédemment, l'agent sédentaire réajuste le *Timeout* du protocole qui est armé dès la réception du message *Entrée()*. Donc, si le *Timeout* expire avant la réception du message *Sortie()*, alors l'agent sédentaire marque cette plate-forme comme attaquante selon une ré-exécution. Il continue à attendre le message *Sortie()* pendant un autre intervalle de temps (*IAS*). Si une autre fois de plus le message n'arrive pas, alors l'*AS* détectera une attaque de déni de service (cf. Figure 3.4). Avant de commencer la prochaine étape de coopération, l'*AS* fait un résumé des messages *Arrive()*, *Entrée()* et *Sortie()*, produit un nouveau message *majASR()* et l'envoie à l'*ASR* pour le mettre au courant de l'exécution de l'*AM* dans la plate-forme P_i ainsi que ses échanges avec ce dernier.

Le message *majASR()*, dont la structure est présentée à la Figure 3.5, contient l'identité de l'*AS* signée par la clé privée de *T*, l'identité de l'*AM*, les identités des plates-formes précédentes ($Prec_i$), courante (IdP_i) et suivante ($Suiv_i$) ; en plus des Entrées (*X*) et Résultats (*R*), il contient également un vecteur (*Attaques*) des attaques éventuelles détectées par l'*AS* lors du saut courant.

De son côté, en recevant le message *majASR()*, l'*ASR* enregistre les données reçues, arme son temporisateur *TimeoutAS* pour guetter une éventuelle panne de l'agent sédentaire lors du prochain saut de l'agent mobile pour prendre la relève de la coopération avec l'*AM*. Le *TimeoutAS* est estimé par l'*ASR* au début du protocole par la même méthode qu'utilise le protocole de base pour estimer son *Timeout*. En effet, avant sa migration vers la plate-forme *T*, l'*AS* envoie d'abord un message *majASR()* de test à l'*ASR* qui est déjà dans la plate-forme *TR*. Puis, avec l'*ASR*, il simule un saut de l'agent mobile ainsi que les messages de coopération. Ensuite, il envoie un message *majASR()* vers l'*ASR*. Ainsi, ce dernier fait une estimation de l'inter-arrivée des deux messages

majASR() entre deux sauts, à laquelle il ajoute la valeur de l'*IAS*. De ce fait, le protocole évitera à l'*ASR* de déclencher la procédure de relève pendant que l'*AS* est toujours actif en attente de la communication de l'*AM* et n'a pas déclaré un déni de service (l'*IAS* n'a pas encore expiré).

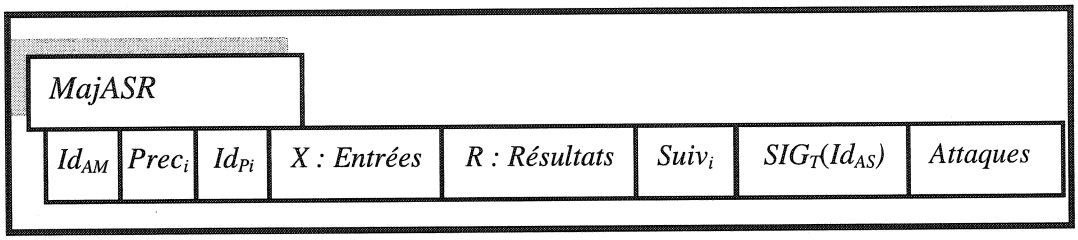

Figure 3.5 Format du message de mise à jour de l'agent sédentaire de reprise

Au cours de l'étape *i*, trois situations particulières peuvent survenir au niveau du protocole : une faute sur la plate-forme *T*, un déni de service et une modification permanente de code. Voyons comment le protocole réagira à chacune de ces situations.

Faute sur la plate-forme T

L'agent sédentaire de reprise ne coopère pas avec l'agent mobile durant ses différentes migrations sauf dans des situations exceptionnelles de pannes. Il interagit uniquement avec l'agent sédentaire coopérant principal *AS*. En fait, le mécanisme fonctionne de la manière suivante : l'*AS* fait propager, vers l'agent sédentaire de reprise, les informations qu'il reçoit à l'issue des messages échangés avec l'agent mobile (*Arrive()*, *Entrée()* et *Sortie()*) via le message *majASR()*. À la réception du message *majASR()*, l'*ASR* arme son temporisateur à la valeur de *TimeoutAS*. Ainsi, si l'agent sédentaire n'envoie plus de message alors que le protocole est toujours opérationnel et le *TimeoutAS* a expiré, l'*ASR* suppose alors la panne de l'*AS*. Par conséquent, il envoie un message *changerCooperant()* (cf. Figure 3.6) de changement de coopérant à l'agent mobile qui était en attente de communication avec son agent coopérant principal, en l'occurrence l'*AS*. Aussi, l'agent mobile détient-il l'adresse de localisation de l'*ASR et*

c'est ainsi qu'il peut utiliser la solution de rechange et s'adresser à l'agent sédentaire de reprise qui est maintenant actif. La coopération est reprise et continue selon le même protocole précédemment décrit. L'agent mobile reprend l'envoi du dernier message (*Arrive()*, *Entrée()* et *Sortie()*) qui précède la détection de la rupture de communication avec son agent coopérant. L'agent sédentaire de reprise prend ainsi la relève.

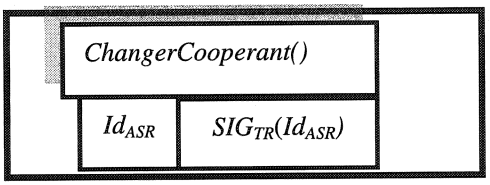

Figure 3.6 Format du message de changement de coopérant envoyé par l'*ASR* à l'*AM*.

En fait, le message *changerCooperant()* est comme un message « hello » de la part de l'*ASR* pour indiquer à l'agent mobile que désormais c'est le deuxième coopérant qui est actif et que l'*AM* doit reprendre les messages dès le début. Ainsi, il pourrait effectuer la coopération en bonne et due forme. Il ne contient que l'identité de l'*ASR* et puis la signature de la plate-forme *TR* sur l'Id_{ASR}. L'aspect sécurité est assuré du fait que seule la plate-forme *TR* aurait cette signature.

La faute sur la plate-forme de confiance *T* survient suite à une panne de la plate-forme elle-même, ou à une panne peu probable de l'agent *AS* sur *T*. En aucun cas cela ne peut être causé par une attaque de *T* contre l'*AS* puisqu'elle est supposée fiable et ne collabore avec aucune plate-forme malicieuse sur l'itinéraire de l'agent mobile. La Figure 3.7 montre la reprise automatique du protocole suite à une faute atteignant l'agent sédentaire *AS* sur la plate-forme de confiance *T*. Ceci se traduit par l'absence de coopération entre l'*AM* et l'*AS* qui signifie un arrêt du protocole. Donc, un basculement du protocole s'effectue pour établir la coopération, mais cette fois-ci, c'est avec l'agent sédentaire de reprise *ASR*.

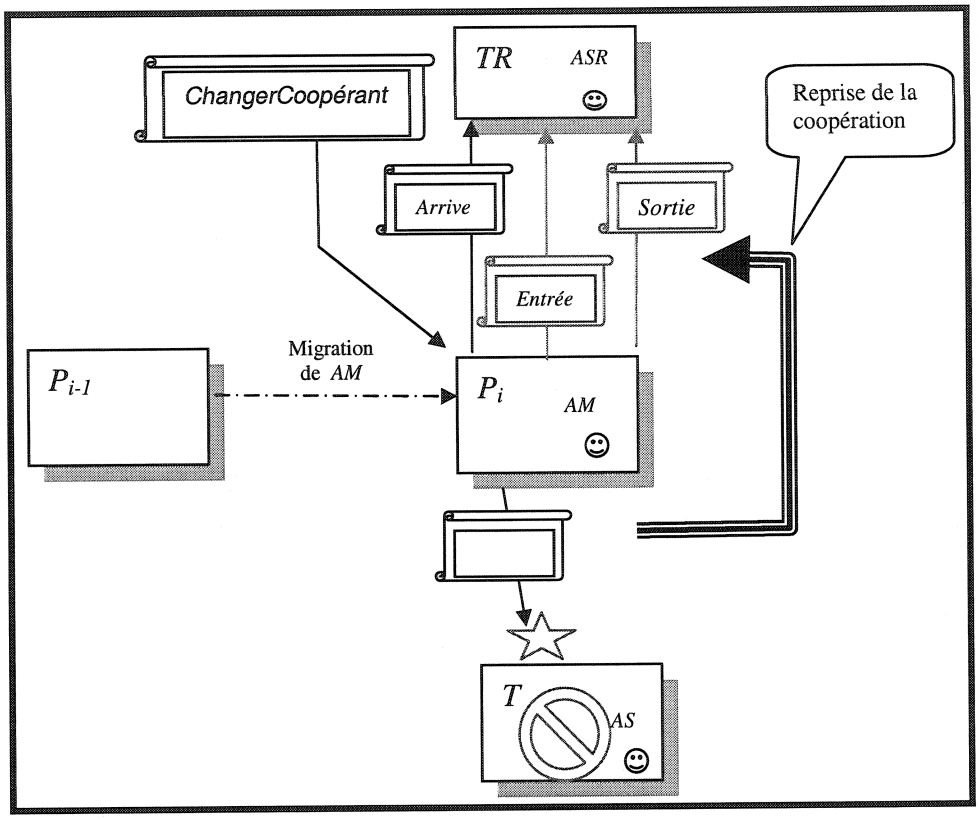

Figure 3.7 Reprise immédiate par basculement automatique après une faute sur la plate-forme de confiance *T*

Déni de service

Un déni de service survient quand, pour une raison ou pour une autre, l'agent mobile ne peut plus répondre. Alors, pour ne pas redémarrer le protocole dès le début, chose que fait d'ailleurs le protocole de base, l'*AS* ou éventuellement l'*ASR* crée une copie de l'*AM* mise à jour avec toutes les informations qu'avait l'ancien agent mobile. C'est à cette fin que sert l'envoi de l'état de l'agent mobile. Ainsi sa nouvelle copie pourrait reprendre là où la copie « défunte » a été arrêtée. Ensuite, il l'envoie vers une plate-forme alternative prévue à cet effet. En fait, à la phase d'initialisation, on initialise aussi une liste de plates-formes alternatives en cas d'attaque de déni de service contre

l'agent mobile. Une fois sur la nouvelle plate-forme, le nouvel *AM* pourrait poursuivre la mission.

Modification permanente de code

Une modification de code en permanence est soupçonnée quand l'*AS* détecte deux attaques successives de modification simple (sur les deux plates-formes P_i et P_{i+1}). Dans ce cas, il va d'abord éliminer l'agent mobile sur la plate-forme P_{i+2} du moment que l'agent mobile y est déjà rendu. Puis, il va envoyer un clone valide et sain vers la plate-forme précédente (P_{i+1}). Ensuite, il va attendre les messages de coopération de la part du nouvel agent mobile. À la réception du message *Sortie()*, il va comparer les résultats reçus avec ceux qu'il a obtenus. Si, cette fois aussi, il reçoit des résultats différents, il conclut que P_{i+1} est bien une plate-forme attaquante (cf. Figure 3.8).

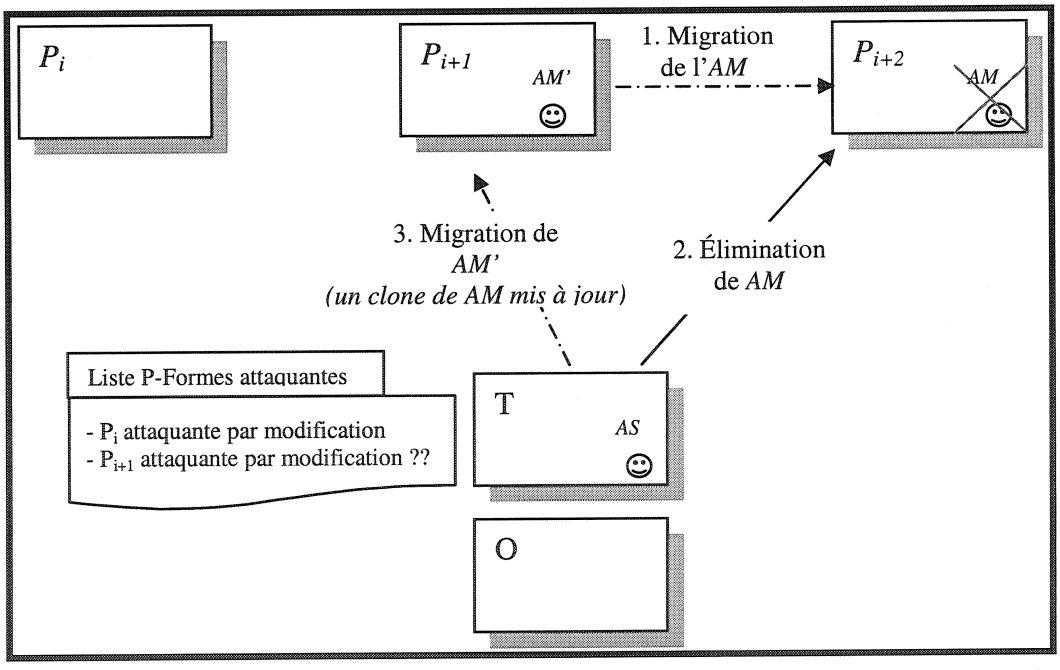

Figure 3.8 Comportement du protocole en présence d'attaque de modification permanente de code

3.2.5 Étape terminale

À la fin du protocole, l'agent mobile rentre à sa plate-forme d'origine. Cette dernière identifie lequel des agents sédentaires coopérait avec l'*AM* l'étape précédente, elle lui demande d'envoyer les résultas intermédiaires ainsi que la liste des plates-formes attaquantes. La plate-forme *O* confronte les deux rapports pour éliminer les résultats des plates-formes identifiées comme attaquantes. Celles-ci auront leur crédibilité mise en cause et seront évitées lors des prochaines sollicitations.

3.3 Analyse de la sécurité du protocole

Le nouveau protocole proposé a pour objectif sinon de protéger l'agent mobile contre les attaques qui peuvent menacer sa sécurité, du moins de détecter celles-ci. Dans ce qui suit, nous allons présenter la manière dont notre protocole réagit face à d'éventuelles attaques. En fait, nous allons simuler différentes attaques et enregistrer le comportement du protocole pour les détecter et les éviter. Nous tenons à préciser que notre protocole permet, en plus de la détection ou la protection assurée par le protocole de référence, de raffiner la détection de certaines attaques et neutraliser l'effet pour d'autres. Plus précisément, nous allons étudier les scénarios d'attaques suivants :

1. modification simple de code ;
2. modification permanente de code ;
3. déni de service ;
4. ré-exécution du code de l'agent mobile.

3.3.1 Attaque par modification simple de code

Toute attaque de manipulation de code fait partie de l'un des scénarios présentés à l'arbre de la Figure 3.9. Ces attaques sont au nombre de sept et sont étudiées dans [ELR02] à savoir :

1- modification du message *Sortie()* seulement ;

2- modification du message code critique seulement ;

3- modification du code critique et du message *Sortie()* ;

4- modification du message *Entrée()* seulement ;

5- modification des messages *Entrée()* et *Sortie()* ;

6- modification du message *Entrée()* et code critique ;

7- modification du message *Entrée()*, code critique et du message *Sortie()*.

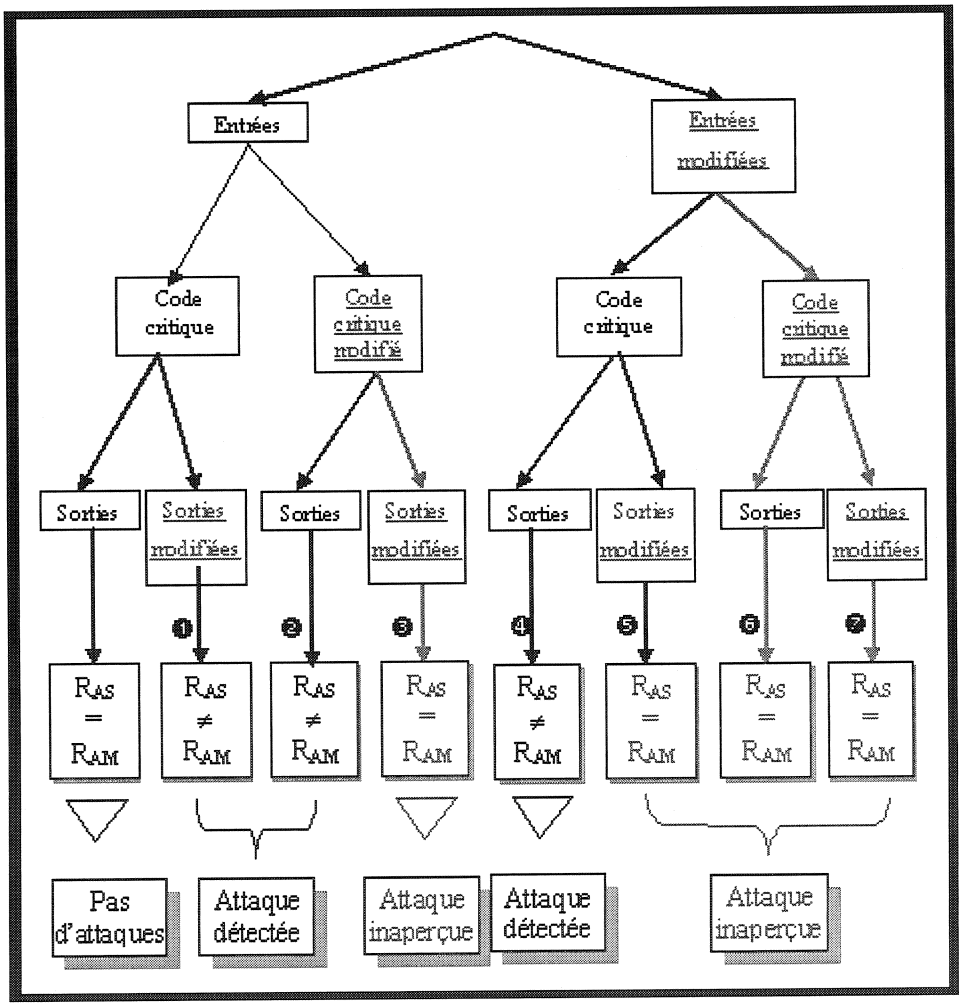

Figure 3.9 Arbre présentant les différents scénarios d'attaques de modification de code

3.3.2 Attaque par modification permanente de code

Deux scénarios sont possibles dans ce type d'attaque que la plate-forme malveillante commet :

1. *avant la fin de l'exécution* de l'agent mobile sur cette plate-forme, auquel cas nous rejoignons l'un des cas des attaques de modification simple ;
2. *après la fin de l'exécution* de l'agent mobile sur cette plate-forme : Dans ce cas, la plate-forme attaquante P_i modifie le code en permanence de telle façon que toutes les plates-formes qui se trouveront ultérieurement sur l'itinéraire de l'agent mobile aient des résultats erronés. Le protocole soupçonnera la plate-forme P_{i+1} comme attaquante, alors qu'elle est peut-être bienveillante. En déclenchant la contre-mesure, si elle est effectivement bienveillante, elle donnera des résultats corrects lors de l'exécution du clone valide de l'*AM*. Ainsi, le protocole identifiera uniquement la plate-forme P_i comme attaquante. Cependant, si P_{i+1}, étant collaborante de P_i, exécute incorrectement le clone valide, alors le protocole marquera P_i et P_{i+1} comme attaquante de modification simple de code, bien que P_i ait commis une attaque permanente et elle a passé en partie inaperçue. Si par contre P_{i+1} a commis une attaque simple contre l'*AM* original et ne la commet pas sur le clone valide, dans ce cas , seule P_i sera marquée et non pas P_{i+1}.

3.3.3 Déni de service

C'est une attaque qu'on ne peut éviter à l'agent mobile d'en être la victime. La seule et meilleure chose qu'un protocole ou une approche puisse faire, c'est d'y survivre avec le moins de perte possible. Dans ce contexte, l'élimination de l'agent mobile par la plate-forme actuelle peut se faire aux moments suivants :

- avant la fin de l'exécution ;
- après exécution : capture ou refus de migration.

Dans le premier cas, nous détectons cette attaque avec le compteur de temps utilisé (temps d'attente supplémentaire décrit précédemment). Une fois détectée, l'agent sédentaire se rend compte que cette plate-forme est attaquante et la marque ainsi. Puis, il crée un clone de l'agent mobile doté de toutes les données qu'avait l'agent mobile avant

sa dernière migration. Ensuite, il l'envoie sur une des plates-formes choisies parmi une liste de plates-formes alternatives prévue à cette fin. En fait, cette liste est constituée avant le lancement du protocole; elle contient des plates-formes qui offrent les mêmes services auxquels l'agent mobile aspire et qui n'ont présenté aucun signe d'hostilité auparavant. Ainsi, dans ce cas, la continuité du protocole est assurée, ce qui évite toute interruption de la mission de l'agent mobile. La Figure 3.10 illustre le comportement du protocole dans des circonstances pareilles.

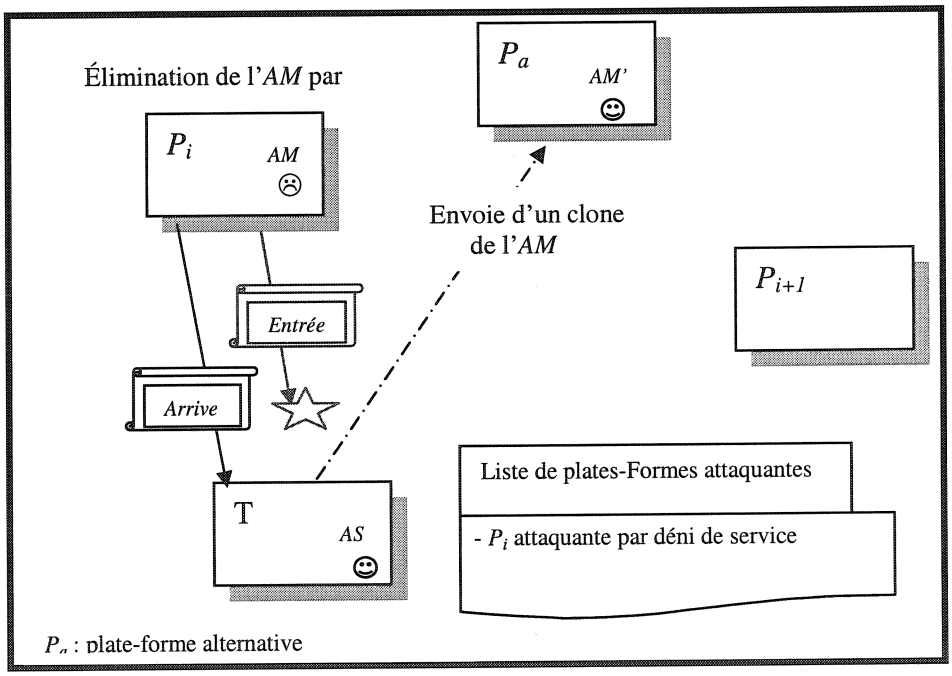

Figure 3.10 Reprise de la mission de l'*AM* suite à une élimination avant la fin de l'exécution

Dans le deuxième cas, l'agent sédentaire détecte cette attaque en l'absence du message *Arrive()*. L'agent mobile a en fait terminé son exécution sur la plate-forme P_i et a manifesté sa volonté de migrer vers la plate-forme suivante P_{i+1}, sauf qu'il n'a jamais pu migrer. Dans ce cas, et comme dans le scénario précédent, l'agent sédentaire crée un clone de l'agent mobile avec toutes les données qu'avait ce dernier avec P_i, sauf que

cette fois il ne l'envoie pas vers une plate-forme au choix, mais vers la plate-forme P_{i+1} où l'agent mobile avait l'intention de migrer (cf. Figure 3.11).

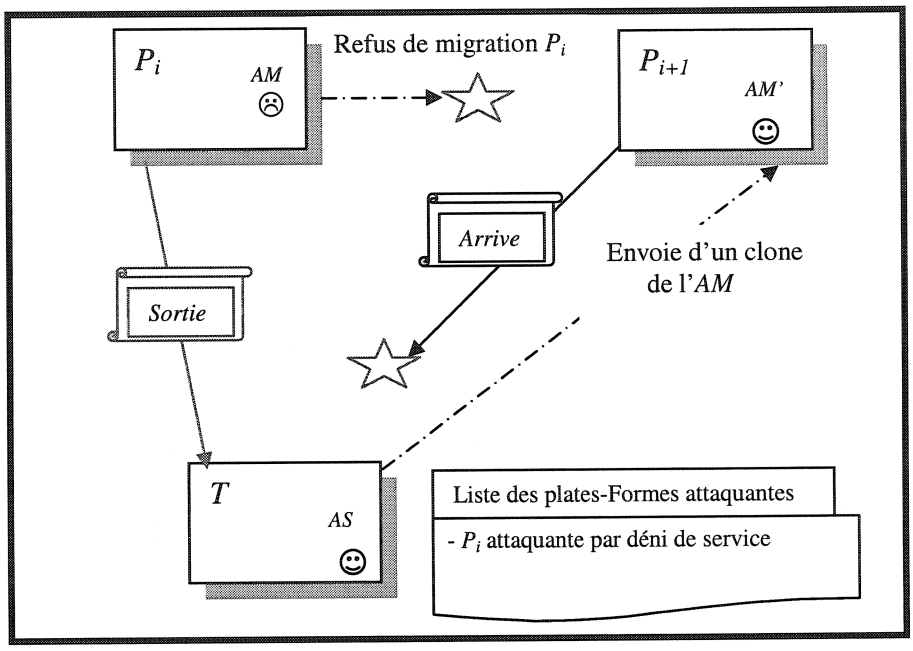

Figure 3.11 Reprise de la mission de l'*AM* suite à une élimination après exécution

3.3.4 Ré-exécution de code

Le protocole détecte une ré-exécution du code grâce au compteur *Timeout*. Dans le protocole de base, ce compteur est basé sur une estimation faite au préalable. Dans notre cas, l'ajustement de ce compteur est fait dynamiquement suite aux messages échangés entre l'agent mobile et l'agent sédentaire. Bien sûr, il tient compte de l'état du réseau, des caractéristiques de la plate-forme actuelle qui exécute l'agent mobile. Cela nous permet d'avoir une valeur plus réelle et plus raffinée, qui diffère d'une plate-forme à une autre, de là, un intervalle d'exécution plus adéquat : ni trop court pouvant ainsi mettre en cause indûment l'intégrité de certaines plates-formes bienveillantes moins performante, ni trop long permettant ainsi de ne pas détecter des plates-formes malveillantes très performantes. Mais, c'est un intervalle qui se veut à la fois convenablement court pour

détecter des plates-formes très rapides, et assez long pour laisser plus de temps à des plates-formes moins performantes. Ajoutons aussi que cette nouvelle estimation n'est guère basée sur des informations recueillies par l'*AM*, mais elle se base juste sur les inter-arrivés des messages de coopération entre l'*AM* et l'*AS*. Donc, une plate-forme qui veut surpasser cette approche de détection ne peut pas savoir ce qui se passe du côté de l'agent coopérant, car c'est ce dernier qui fait les relevés du temps et estime le *Timeout*. Ainsi, toute attaque de ce genre de la part d'une plate-forme malicieuse ne peut passer inaperçue.

3.4 Analyse du protocole en présence des autres attaques

Dans cette section, nous allons passer en revue toutes les attaques citées dans le chapitre 2 pour expliquer le comportement de notre protocole face à chacune d'elles. Nous allons aussi proposer les solutions éventuelles au cas ou notre protocole ne prévoit pas de contre-mesures.

1. Espionnage du code, espionnage des données, espionnage de l'état : la plate-forme qui exécute l'agent mobile détient en fait toutes ces ressources. Elle les manipule volontairement. Donc pour empêcher un éventuel espionnage, le concepteur pourrait utiliser les techniques d'*algorithmes de confusion* ou la *boîte noire* [HOH98] pour dissimuler le code de l'*AM* et en complexer la compréhension. En ce qui concerne les données, la plate-forme qui exécute l'*AM* a accès à tout. Alors, notre proposition serait d'interdire au moins l'accès aux plates-formes ultérieures à l'aide de *journaux à ajout seule* utilisés dans le système *Ajanta* [KAR00].

2. Espionnage des interactions avec les autres agents : les messages sont chiffrés par les clés publiques des plates-formes qui abritent les agents en question, donc aucune tierce ne peut espionner ce message sauf bien sûr la plate-forme qui abrite l'agent mobile car celle-ci peut l'espionner avant l'opération de chiffrement.

3. Vol, piratage et clonage : ces attaques consistent à faire des copies illégales de parties ou totalité de l'agent mobile. Ce problème rejoint celui que vivent tous les

producteurs de logiciels ou de musiques à travers la planète. Il n'y pas de procédure de protection contre le piratage et le clonage. Néanmoins, nous devrions rendre au pirate la tâche difficile d'exploiter ce qu'il a volé. Car, pour la plupart, ces attaques ne constituent pas des attaques en elles-mêmes, mais elles préparent d'autres attaques plus dangereuses. Pour ce faire, on pourrait utiliser soit les *algorithmes de confusion* ou le *modèle de boite noire limitée dans le temps* [HOH98].

4. La rétro-ingénierie (reverse engineering) : cette attaque demande beaucoup d'ingéniosité de la part de l'attaquant et n'aurait d'effet que si le même agent mobile effectue fréquemment des sauts dans cette plate-forme.

5. Manipulation du code (modification permanente) : pour pallier la lacune du protocole, nous proposons l'utilisation de l'approche des *traces d'exécution* [VIG98]. Ainsi, si la plate-forme attaque la 1ère fois et non pas la 2ème fois, elle ne sera pas détectée par l'*AS*, mais la plate-forme d'origine la détectera au retour de l'*AM*.

6. La manipulation des données, la manipulation de flux de contrôle ou état et la manipulation des interactions avec les autres agents (voir Figure 3.9) : ces attaques consistent généralement en une combinaison d'attaques de modification contre les données (entrées), le code ou les résultats (sorties) à savoir : (code, résultats), (données, résultats), (données, code) et (données, code, résultats). Pour détecter ces attaques, on pourrait utiliser des *traces d'exécution* de *Vigna et al.* [VIG98] comme pour le cas de la manipulation permanente de code. Cette approche ne permet ni de protéger l'agent contres ces attaques ni de les détecter en temps réel mais néanmoins elle permet d'identifier l'attaque au retour de l'agent mobile.

7. La mascarade de la plate-forme : elle est souvent perpétrée contre un agent en vue d'un changement d'itinéraire entre deux plates-formes qui collaborent et c'est une conséquence de l'attaque « retour de résultats erronés des appels systèmes effectués par l'agent dans cette plate-forme ». Cette attaque est immédiatement détectée par l'agent coopérant sauf si la plate-forme P_i collabore avec une autre plate-forme P_j , illégalement introduite dans l'itinéraire de l'*AM*. En effet, P_i dévie l'*AM* de son itinéraire

en l'envoyant vers P_j au lieu de P_{i+1}, alors le protocole ne pourrait pas la détecter. Mais tout autre scénario (attaque sans collaboration avec une autre plate-forme) est détecté.

8. Le déni d'exécution et le refus de migration : ce sont deux facettes du déni de service.L'approche proposée constitue une mesure de protection contre les dénis de service. Certes, nous ne pouvons pas l'éviter, mais nous pouvons au moins assurer la survie de notre protocole après une attaque de ce genre.

9. Le délai d'exécution : en premier lieu, cette attaque est détectée par la contre-mesure de ré-exécution de code. Cependant, si ce délai est prolongé, alors c'est la contre-mesure de déni de service qui sera déclenchée.

3.5 Synthèse du protocole proposé

Le nouveau protocole se propose d'apporter des améliorations au protocole de base [ELR02]. Nous nous sommes fixé comme objectif le développement de la mesure de détection des attaques de ré-exécution de code, de déni de service et de la modification permanente de code afin de rendre le protocole tolérant aux fautes. En effet, la détection de la ré-exécution de code se base sur une méthode dynamique d'estimation du *Timeout*. L'approche ne laisse aucune chance à une plate-forme malveillante de perpétrer une attaque de ré-exécution sans se faire déceler. Aussi, non seulement elle détecte le déni de service, mais aussi, elle permet au protocole de survivre après une telle attaque. On pourrait même la considérer indirectement comme une mesure de détection et de protection.

Quant à l'attaque de modification permanente de code, nous avons proposé une solution qui détecte en partie cette menace, en plus de redresser l'agent mobile et permettre sa pérennité. En ce qui concerne la robustesse, le protocole permet la reprise en cas de panne d'une de ses entités, ce qui permet d'éviter à l'agent mobile plusieurs parcours inutiles. Aussi, le nouveau protocole invalide le besoin d'une entité appelée agent estimateur puisqu'en partie ce sont les agents du protocole qui font les estimations nécessaires.

Cependant, la proposition entraîne certainement quelques contraintes comme l'ajout d'une deuxième plate-forme de confiance, qui constitue en général un service payant pouvant éventuellement augmenter le coût. Aussi, le fait de rendre le protocole tolérant aux fautes nous amène à se demander à quel prix ceci est possible. Bien sûr, les messages de mise à jour ajoutent bien du trafic au réseau. De plus, les procédures de reprise après les fautes sont coûteuses en terme de temps, ressources mémoire et trafic réseau. En effet, elles nécessitent de générer un clone valide de l'agent mobile dans le cas du déni de service, ou encore d'éliminer l'*AM* et de générer son clone en cas de modification permanente de code, ou finalement de retransmettre certains messages en cas de panne de l'*AS*.

CHAPITRE IV

VÉRIFICATION FORMELLE DU PROTOCOLE, IMPLÉMENTATION, TESTS ET RÉSULTATS

La suite logique des choses nous amène à vérifier et tester notre protocole déjà conçu au chapitre précédent. Tout d'abord, nous allons nous assurer qu'il répond bien aux spécifications, c'est une étape qui complète et qui valide en fait la phase de conception. Ensuite, nous allons l'implémenter pour effectuer certains tests afin d'en évaluer les performances. Pour ce faire, nous allons vérifier formellement notre protocole en trois étapes : la modélisation de notre protocole en utilisant un des modèles les plus adéquats; la simulation pour valider les fonctionnalités du protocole et la vérification par la technique du *model-checking* des propriétés que notre protocole possède. Et pour tester le protocole, nous allons l'implémenter sur une plate-forme d'agents mobiles pour pouvoir le comparer au protocole de base d'une part et d'autre part pour en tester les nouvelles fonctionnalités.

4.1 Vérification formelle du protocole

La vérification formelle d'un logiciel ou d'un protocole est indispensable pour en assurer la fiabilité, la cohérence des fonctionnalités et la certitude que le comportement du protocole ou logiciel répond bien aux attentes généralement décrites dans les spécifications. La technique de vérification basée sur le *model-checking* consiste à modéliser le protocole par un ou plusieurs systèmes de transitions appelés automates, sur lesquels les propriétés attendues, exprimées en logique temporelle, sont vérifiées à l'aide d'outils appelés *model-checkers*. La vérification consiste en la preuve formelle des propriétés structurelles et comportementales que l'on en attend.

4.1.1 Quelques définitions et notations

Un automate temporisé est constitué d'un ensemble de sommets et de transitions. En plus de leurs noms, les places possèdent ou non des invariants : ce sont des expressions booléennes qui consistent en une combinaison des horloges et/ou des variables de l'automate et que le système doit satisfaire pour pouvoir rester dans cette place. De leur côté, les transitions possèdent trois éléments : les conditions de franchissement (*Guard*), la synchronisation et les mises à jour. La condition de franchissement devrait être vraie pour pouvoir franchir la transition. La synchronisation exprime, quant à elle, l'action de la transition. Elle se fait par rendez-vous d'une action *a!* d'un automate (similaire à un envoi de message, le signe « ! » apparaît à la fin de l'action) avec une action *a?* d'un autre automate (similaire à un envoi de message, le signe « ? » apparaît à la fin de l'action). Quand la transaction est franchie, des variables et/ou des horloges locales ou partagées peuvent être mises à jour (ex. initialiser une horloge).

Nous introduirons aussi les définitions suivantes :

- *Réseau d'automates temporisés* : c'est un ensemble d'automates représentant les différents processus d'un seul système. Ces processus sont synchronisés par l'échange de messages ou des variables partagées. Et pour représenter l'aspect temporel, ces automates sont munis d'horloges permettant de mesurer et de contrôler le temps écoulé entre les actions.
- *Propriété formelle* : c'est une formule, exprimée selon la syntaxe d'un langage de spécification formelle, qui représente un comportement ou une fonctionnalité que possède le système.

Pour nommer les places de l'ensemble du réseau d'automates, nous avons adopté la procédure de notation suivante : les noms des places de chaque automate sont préfixés du nom du processus. Par exemple, la place début de l'automate de l'agent *AS* sera notée *AS_Debut*

4.1.2 Environnement de validation

Notre protocole est constitué de processus qui interagissent entre eux. Ces derniers passent par un ensemble d'états qui représentent le fonctionnement des composantes du système. À chaque instant, le système se trouve dans un état caractérisé par un vecteur qui donne la place où se trouve chaque processus ainsi que les valeurs des horloges et des variables. De ce fait, une modélisation à l'aide d'automates sera plus adéquate. Par ailleurs, le choix de l'outil de vérification est beaucoup influencé par les spécificités de notre protocole. Vu que nous devons vérifier des propriétés pour lesquelles l'aspect temporel est déterminant, nous avons écarté le model-checker SPIN[1] utilisé pour la validation du protocole de base [ELR02] du fait qu'il ne permet pas de modéliser le temps. Cependant, nous avons choisi un outil plus convivial qui modélise l'aspect temporel et en plus il permet de vérifier des réseaux d'automates temporisés que nous avons adoptés pour modéliser notre protocole : C'est l'outil UPPAAL [UPP01] [UPP02], un produit universitaire disponible gratuitement sur l'Internet.

UPPAAL est un model-checker qui a été développé en collaboration par le Groupe de conception et d'analyse des systèmes temps réels (Design and Analysis Of Real Time System Group) de l'université Uppsala en Suède et le groupe des recherches fondamentales en génie informatique (Basic Research in Computer Science) de l'université Aalborg au Danemark. Il se veut un environnement intégré pour la modélisation, la simulation et la vérification des systèmes temps réels. Ces derniers peuvent être modélisés par des collections de processus munis d'horloges réelles et communiquant à l'aide de canaux ou messages et des variables partagées. UPPAAL comprend trois parties : un langage de spécification qui peut décrire des réseaux d'automates temporisés, un simulateur qui permet d'examiner toutes les exécutions possibles dès les premières phases de conception, et un model-checker qui effectue une recherche exhaustive couvrant ainsi tous les comportements dynamiques et possibles du réseau d'automates. Cet ensemble de composants fonctionne en mode client/serveur : un

[1] http://spinroot.com/spin/

client java (une interface graphique d'utilisateur consistant en un éditeur, un simulateur et un vérificateur) peut communiquer, par le biais d'un protocole basé sur des sockets, avec le serveur qui est en fait le moteur du model-checker.

En ce qui concerne la logique temporelle, nous avons utilisé CTL (Computational Tree Logic). Ainsi, il nous a été possible de manipuler des horloges. Cette logique est supportée en grande partie par UPPAAL.

4.1.3 Modélisation du système

Notre protocole se compose de quatre acteurs principaux, à savoir : l'agent mobile, l'agent sédentaire, l'agent sédentaire de reprise et la plate-forme d'origine. Tout d'abord, nous allons identifier ces acteurs à des processus. Ensuite, pour modéliser le fonctionnement global du protocole, nous représenterons chacun des processus par un automate temporisé. Nous allons adopter la notation suivante : l'agent mobile *AM()*; l'agent sédentaire *AS()*; l'agent sédentaire de reprise *ASR()*, et la plate-forme d'origine *PO()*.

Pour avoir une meilleure vérification du protocole, nous avons modélisé un processus externe au protocole qui pourrait générer des attaques et des scénarios particuliers comme les pannes. Ainsi, notre modélisation se rapprochera plus de la réalité; sinon, nous serons dans l'obligation de modifier les modèles des processus afin d'obtenir les comportements escomptés. Cependant, si de telles modifications atteindront les objectifs en terme de comportement, alors les modèles ainsi obtenus seront loin des processus réels. Nous notons ainsi *PA()* le processus qui représente une plate-forme attaquante. Voici la description des processus :

- ***Le processus PO()*** (c.f. Figure A.1) : C'est le processus responsable de déclencher le protocole. Il lance les autres processus. L'automate de la Figure A-1 (Annexe A) décrit le fonctionnement du processus *PO()*. À partir de son état *Début*, il envoie séquentiellement les synchronisations suivantes :

Create_AS et *Move_AS* respectivement pour lancer l'*AS()* et le faire déplacer (vers la plate-forme *T*) ;
Create_ASR et *Move_ASR* respectivement pour lancer l'*ASR()* et le faire déplacer (vers la plate-forme *TR*) ;
ActiverPA pour activer la plate-forme attaquante ;
Create_AM, *Move_AM* respectivement pour lancer l'*AM()* et le faire déplacer (vers sa première destination).

À la fin du protocole, *PO()* reçoit la synchronisation *FinMigrations* de la part du processus *AM*() quand la condition qui porte sur la variable *Saut* est satisfaite. En fait, cette variable est utilisée dans le contexte de simulation uniquement pour faire le compte du nombre de plates-formes visitées; ainsi, au bout d'un nombre de sauts, le protocole prendra fin. Puis, *PO()* envoie une synchronisation *FinProtocole* à *AS()* ou *ASR()*, dépendamment de celui qui est actif à l'instant, pour mettre fin à sa mission. Ensuite, il envoie *FinPA* au processus *PA()* comme signe de fin de protocole.

- ***Le processus AS()*** (c.f. Figure A.2) : Comme tous les autres processus, l'*AS()* se trouve au départ dans son état *AS_Debut*. Au lancement du protocole, l'*AS()* reçoit l'action *Create_AS* puis *Move_AS* de *PO()*, il est alors créé et se déplace vers la plate-forme *T*. À l'état *AS_AttenteArrive*, il attend alors que l'*AM()* communique avec lui (un message *Arrive*). À la réception de ce message, l'*AS()* passe alors à la place *AS_AttenteEntree* d'où il peut attendre un maximum de *Timeout* unités de temps avant de recevoir un message *Entree*. Il passe ainsi à l'état *AS_AttenteSortie* en transitant par la place intermédiaire *AS_4*. Par la suite, il peut attendre une autre fois un maximum de *Timeout* unités de temps avant de recevoir un message *Sortie* après quoi il passe aux états *AS_6* puis *AS_7*. À ce niveau, *AS()* utilise une variable *CompRes* qui modélise la comparaison des résultats obtenus par l'agent sédentaire avec ceux envoyés par l'agent mobile. À partir de *AS_7*, si les résultats sont identiques (*CompRes==1*), alors le processus transite vers l'état *AS_ApresSortie*. Sinon, il transite vers l'état *ASR_ResIncorrect*. Encore, s'il se trouve que la variable *AS_MP* (compteur des attaques de modification de code) atteint

la valeur 2, le processus *AS()* détecte une modification de code permanente et de ce fait, il doit soupçonner que l'agent mobile n'est plus valide. Ainsi, il transite à l'état *AS_AgentModifie*. Dans ce cas, il doit mettre fin à l'exécution de l'*AM()* et doit en recréer un clone valide (les place *AS_9* et *AS_AM_Recree*). Si *AS_MP* est inférieur à 2, alors l'*AS()* rejoint la place *AS_ApresSortie* et doit pouvoir envoyer un message *maj_ASR* au processus *ASR()*. Pour pouvoir transiter au début du cycle, l'*AS()* doit recevoir une synchronisation *Sig_Move_AM* de la part du processus *AM()* lui indiquant qu'il est arrivé à la plate-forme suivante et ainsi de suite. Si le protocole prend fin, l'*AS* recevra alors un message *FinProtocole* du processus *PO()*. Il transite alors à son état de départ (*AS_Debut*). Étant donné que l'*AS()* se trouve dans une des places *AS_AttenteEntree* ou *AS_AttenteSortie*, s'il reçoit une synchronisation *RC* de la part de *PA()*, alors il sera en mesure de détecter une ré-exécution de code. Pour cela, il transite respectivement à la place *AS_RCDetectee* ou *AS_RCDetectee2* tout en initialisant l'horloge *h2* qui modélise l'*IAS* permettant de détecter un déni de service. Et si encore une fois il reçoit la synchronisation *DeniService_AC* de la part de *PA()*, il transite vers la place *AS_DoSDetecte*. De là, il envoie un message *maj_ASR* à l'*ASR()*. Et à partir de la place *AS_DoS*, il sera en mesure de créer un nouvel agent mobile *(Create_AM)* pour arriver à l'état *AS_AM_Recree* comme dans le cas de modification permanente de l'agent mobile.

À tout moment, l'*AS()* peut recevoir une synchronisation *Fin_AC* de la part de l'*ASR()*. Ceci survient suite à la réception de ce dernier d'une synchronisation *panne_T* envoyée par le processus *PA()* et qui simule une panne de la plate-forme *T*.

- ***Le processus ASR()*** (c.f. Figure A.3) : Au début, l'*ASR()* se trouve dans son état *ASR_Debut*. Au lancement du protocole, l'*ASR()* reçoit l'action *Create_ASR* puis *Move_ASR*, après quoi il migre vers la plate-forme *TR* comme prévu par le protocole. Dans l'état *ASR_migre*, il attend alors que l'*AS()* communique avec lui. À la réception du message *maj_ASR*, il passe à l'état *ASR_NonActif* et continue à recevoir les messages *maj_ASR* en provenance de l'*AS()*. Il demeure en cet état tant que l'*AS()* est actif (pas de

réception de la synchronisation *Panne_T*). L'*ASR()* passe à l'état *ASR_PanneAS* dès qu'il reçoit la synchronisation *panne_T* du processus *PA()*. Ensuite, il envoie *Fin_AC à l'AS()* pour que ce dernier revient à son état début (*AS_Debut*). Il passe ainsi à l'état *ASR_PanneDetectee* et, pour prendre la relève (état *ASR_AttenteArrive*), il doit envoyer un message *Changement_AC* à l'*AM()* lui indiquant de changer de coopérant. Par la suite, le fonctionnement de l'*ASR()* rejoint celui de l'*AS()* précédemment décrit, sauf les envois de message de *maj_ASR* car l'*ASR()* n'aura pas d'agents qui feront la relève en cas de panne.

- ***Le processus AM()*** (c.f. Figure A.4) : comme les autres processus, tout commence à partir de la place *AM_Debut* où il reçoit les synchronisations *Create_AM* puis *Move_AM*. Se trouvant ainsi à la place *AM_ApresMove* où il peut commencer la coopération avec l'agent sédentaire *AS* qui est actif par défaut, il lui envoie alors un message *Arrive* pour se trouver à l'état *AM_ApresArrive*. En parcours normal, c'est à dire sans fautes ni attaques, l'*AM()* va pouvoir envoyer successivement les synchronisations *Entree* puis *Sortie* pour se trouver successivement aux places *AM_Critique* puis *AM_FinSaut*. Par la suite, il signale à l'*AS()* sa migration vers la plate-forme suivante (*Sig_Move_AM*). Et le cycle recommence jusqu'à la fin du protocole, et là, il envoie une synchronisation *FinMigrations* au processus *PO()* pour marquer la fin de son parcours. Néanmoins, en présence d'attaques de ré-exécution de code, le processus *AM()* transite soit par l'état *AM_RC* à partir de l'état *AM_ApresArrive* ou par *AM_RC2* à partir de l'état *AM_Critique* en recevant dans les deux cas la synchronisation *RC_AM* en provenance du processus *PA()*. En cas de déni de service, le processus *PA()* envoie une synchronisation *DeniService_AM* au processus *AM()* qui, se trouvant à l'un des états *AM_RC*, *AM_RC2* ou *AM_FinSaut,* transitera vers l'état *AM_DS*. À l'état *AM_FinSaut*, l'*AM()* pourrait recevoir une synchronisation *Fin_AM* et transiter vers l'état *AM_AgentModifie* (équivalent à l'état *AS_AgentModifie* du processus *AS()* : quand la variable *AS_MP* est égale à 2 (modification permanente de code)). À partir des états *AM_DS* (déni de service) et *AM_AgentModifie,* l'*AM()* se verra immédiatement envoyé

un message *Create_AM* qui le fera transiter vers l'état *AM_1* et ainsi, il va recevoir une synchronisation *Move_AM* pour reprendre le cycle.

À tout moment le processus *AM()* peut recevoir un message *Changement_AC* en provenance du processus *ASR()* lui indiquant de changer de coopérant et que désormais il doit coopérer avec lui (en envoyant les messages *Arrive*, *Entree*, *Sortie* *Sig_Move_AM*,...).

Dans l'automate de *AM()*, nous employons la variable *etatAM* pour savoir à quelle étape de l'exécution l'*AM()* s'est rendu. Ainsi, le processus *PA()* pourrait générer des attaques de modification en conséquence, car il ne peut pas générer une modification de code alors que l'*AM()* n'a pas encore commencé l'exécution. Quant à la variable *Sig_move*, nous l'utilisons pour synchroniser l'*AM()* avec l'*ASR* quand ce dernier prend la relève pour recevoir ses premiers messages de coopération. Aussi, la variable *Saut*, nous permet de compter le nombre de sauts que fait l'*AM()*; ainsi, nous avons pu contrôler la longueur de l'itinéraire.

- Le processus PA() (c.f. Figure A.5) : Il sert à modéliser une plate-forme qui entreprend des attaques contre l'agent mobile, ou à générer des pannes au niveau de la plate-forme *T*, ceci via la modification des variables partagées que manipulent les processus *AS()*, *ASR()*, *AM()* et *PA()*, ou par envoi de synchronisations. Nous sommes arrivés à modéliser les attaques sous forme de manipulations de variables ou horloges. Ainsi, il nous a été possible de simuler des attaques en temps réel vu que nos processus fonctionnent en réseau d'automates et sans pour autant modifier le comportement des autres processus. Nous générons les attaques de façon aléatoire, et nous pourrions avoir plus d'une attaque lors du saut de l'*AM()*.

Le processus *PA()* possède trois états : Début (*PA_Debut*), Prêt (*PA_Prete*) et attaque (les places suivantes : *PA_MSC, PA_2emeMSC, PA_RC, PA_DS, PA_PanneAS*). Au début du protocole, il est à l'état *PA_Début*. Au lancement du protocole, *PA()* est activé par le processus *PO()* via la synchronisation *ActiverPA*. Il passe alors à l'état *PA_Prete* d'où il peut perpétrer des attaques. Ensuite et aléatoirement, *PA()* passe en état

d'attaque qu'il quitte immédiatement après. Il peut ainsi perpétrer l'une des attaques suivantes:

- Modification simple de code : la condition importante dans ce cas, c'est que le processus *PA()* soit en exécution de code (*etatAM* >0). Le processus *PA()* effectue l'acte d'attaque en mettant la valeur de la variable *CompRes* à 0. Il provoque ainsi une attaque de modification de code. Il passe tout de suite après à l'état *PA_MCS* (pour modification simple de code). Lors de l'examen de cette variable (*CompRes*), l'*AS()* peut déceler qu'il y a eu attaque et se comporter en conséquence. Le *PA()* met aussi à jour d'autres variables comme *PA_ms* (pour dire qu'une première modification a eu lieu) et *PA_passe* (pour que la deuxième attaque de modification successive n'aura lieu que lors du prochain saut de l'*AM()*), toutes les deux utilisées pour pouvoir attaquer par une modification simple et successive de code.
- Modification permanente de code (modifications simples successives) : le *PA()* perpètre cette attaque en transitant vers l'état *PA_2emeMCS* et revient à l'état *PA_Prête.* Il doit satisfaire les conditions de franchissement de la transition à savoir :
    - *PA_ms* doit être égale à 1 pour signifier qu'une autre attaque de modification simple a déjà eu lieu ;
    - *PA_passe* doit satisfaire l'égalité *PA_passe == Saut - 1.* Ceci est pour assurer au *PA()* que l'*AM()* a déjà migré vers sa prochaine destination après avoir été attaqué par une première modification de code simple;
    - *etatAM* qui doit être supérieur à 0 pour être sûr que l'*AM()* est en phase d'exécution.

Il met à jour alors les variables *CompRes* (objet de l'attaque), *PA_ms* pour interdire cette attaque lors du prochain saut.

- Ré-exécution de code : *PA()* génère cette attaque en envoyant une synchronisation *RC_AC* à l'*AS()* et transite alors à l'état *PA_RC.* Il doit satisfaire une seule condition de franchissement à savoir : « *PA_rc* doit être égale à 0 pour que le

processus fasse cette action une seule fois pendant un saut de l'*AM()* ». Puis, il fait les mises à jour suivantes :

- Il simule l'écoulement du temps pour que le compteur de temps *h1* (horloge partagée) au niveau de l'*AS()* atteigne la valeur *Timeout* et expire ainsi ;
- La variable *attaque* est mise à 1. Cette variable est en fait utilisée dans le seul but de pouvoir exprimer une propriété formelle que nous n'avons pas pu spécifier à l'aide de la syntaxe de UPPAAL.

Ensuite, le *PA()* revient immédiatement à l'état *PA_Prête* en envoyant une synchronisation *RC_AM* à l'*AM()* et en mettant la valeur de la variable *PA_rc* à 1 pour qu'un déni de service puisse être généré.

- Déni de service : La détection de déni de service est basée sur un compteur de temps. Vu que la détection d'une ré-exécution de code est basée aussi sur un autre compteur de temps qui est inférieur au premier, alors le détecteur du déni de service ne sera opérationnel que si le compteur chargé de la ré-exécution de code expire. C'est pourquoi la condition « *PA_rc* == 1 » est nécessaire au franchissement de la transition qui a pour action de générer le déni de service. Pour ce faire, le processus *PA()* envoie la synchronisation *DeniService_AC* à l'*AS()* (éventuellement à l'*ASR()* s'il est actif). Comme dans la ré-exécution de code, il simule l'écoulement de temps en mettant la valeur de l'horloge *h2* (compteur de temps pour le déni de service) à *IAS*; ainsi, le temporisateur de l'*AS()* expire. *PA()* atteint alors l'état *PA_DS* et revient immédiatement à l'état *PA_Prete* en envoyant une synchronisation *DeniService_AM* à l'*AM()* et en remettant la variable *PA_rc* à 0 pour que l'attaque de ré-exécution de code puisse être générée dans le futur.
- Panne de la plate-forme *T* : En fait, la panne de la plate-forme *T* n'est pas une attaque. Cependant, nous l'avons incluse dans ce processus pour ne pas avoir à construire un automate à part entière qui n'accomplira que cette tâche. Pour générer

cette panne, *PA()* doit atteindre l'état *PA_PanneAS*, et doit vérifier, au préalable, les conditions de franchissement de cette transition, à savoir :

- « *etatAM* supérieur à 0 » pour s'assurer que l'*AM()* est en exécution sur l'une des plates-formes et non pas entrain de migrer ou en présence de déni de service ;
- « *AS_MP* inférieur à 2 » pour éviter que le modèle soit en présence d'une modification permanente de code, auquel cas l'*AS()* devrait éliminer l'*AM()* courant, en créer un clone valide et le faire migrer vers la plate-forme précédente dans son itinéraire.

Une fois la transition franchie, *PA()* envoie la synchronisation *Panne_T* à l'*ASR(),* qui modélise l'agent sédentaire de reprise, pour prendre la relève. Pour les mises à jour, le *PA()* effectue les modifications suivantes :

- *h3* est remise à *TimeoutAS*, cela simule en fait l'écoulement de temps. En fait, cet horloge modélise le compteur de temps au niveau de l'*ASR()* afin de détecter une éventuelle panne de *AS()*;
- *Sig_Move* est remise à zéro pour éviter que l'*AM()* quitte la plate-forme actuelle alors qu'il y a une panne de l'*AS()* ;
- *AC_actif* est remise à 2 pour indiquer que c'est le deuxième agent sédentaire (*ASR*) qui est actif ; *panneAS* est remise à 1, cette variable est utilisée dans le but de pouvoir exprimer une propriété formelle que nous ne pouvons pas spécifier avec la syntaxe de UPAAL (similaire à la variable *attaque*).

4.1.4 Propriétés à vérifier

Les propriétés à vérifier sont en général des propriétés de vivacité, c'est à dire qu'un état bien déterminé doit être atteint à partir d'un autre état si des conditions sont satisfaites [BER99]. Cela nous assure que le protocole réagit bien dans certaines situations d'attaques. Ces propriétés de vivacité sont plus fortes que celles de l'atteignabilité qui ne stipulent qu'une possibilité d'atteindre l'état et non pas une certitude. À côté de ces propriétés de vivacité, nous devrons certainement spécifier les

propriétés qui assurent la disponibilité et la survivabilité de notre protocole, deux propriétés qui font partie des politiques de sécurité des systèmes. En utilisant la logique temporelle CTL, nous avons pu spécifier les propriétés suivantes que nous avons regroupées en trois catégories :

- Propriétés de vivacité

Dans cette catégorie, nous citons les propriétés suivantes :

1- Détection de la ré-exécution de code :

Il est toujours vrai que si, à l'état *AS_AttenteEntree* respectivement *AS_AttentSortie*, le *Timeout* expire, on atteint fatalement l'état *AS_RCDetectee* respectivement *AS_RCDetectee2 :*

- *A[](AS.AS_AttenteEntree and h1==Timeout imply A<> AS.AS_RCDetectee)*
- *A[](AS.AS_AttenteSortie and h1==Timeout imply A<>AS.AS_RCDetectee2)*

que nous écrivons dans UPPAAL de la manière suivante:

- ***AS.AS_AttenteEntree and h1==Timeout -->AS.AS_RCDetectee***
- ***AS.AS_AttenteSortie and h1 == Timeout --> AS.AS_RCDetectee2***

(De la même manière, nous pouvons définir ces mêmes propriétés dans le cas de l'*ASR()*).

2- Détection du déni de service :

Il est toujours vrai que si, à l'état *AS_ApresSortie ou AS_RCDetectee* ou *AS_RCDetectee2*, le *IAS* expire, on atteint fatalement l'état *AS_DoSDetecte :*

A[]((AS.AS_ApresSortie or AS.AS_RCDetectee or AS.AS_RCDetectee2)

and h2 == IAS imply A<>AS.AS_DoSDetecte)

que nous écrivons dans UPPAAL de la manière suivante:

(AS.AS_ApresSortie or AS.AS_RCDetectee or AS.AS_RCDetectee2) and

h2 == IAS --> AS.AS_DoSDetecte

(De la même manière, nous pouvons définir cette propriété dans le cas de l'*ASR()*).

3- La génération d'attaque entraîne la détection :

La propriété se lit « Il est toujours vrai que si une attaque est générée par le *PA* fatalement, elle sera détectée soit par l'*AS* ou l'*ASR* » :

- *A[](PA.PA_RC imply A<> (AS.AS_RCDetectee or AS.AS_RCDetectee2 or ASR.ASR_RCDetectee or ASR.ASR_RCDetectee2))*
- *A[](PA.PA_DS imply A<>(AS.AS_DoSDetecte or ASR.ASR_DoSDetecte))*

Nous les écrivons dans UPPAAL de la manière suivante:

- ***PA.PA_RC --> (AS.AS_RCDetectee or AS.AS_RCDetectee2 or ASR.ASR_RCDetectee or ASR.ASR_RCDetectee2)***
- ***PA.PA_DS --> (AS.AS_DoSDetecte or ASR.ASR_DoSDetecte)***

- Propriétés de disponibilité et de survivabilité

Dans cette classe, nous énumérons les propriétés suivantes :

1- Le protocole ne se bloque jamais :

Dans la logique CTL, la propriété s'écrit ainsi : *A[](EX)*. Dans UPPAAL, elle est spécifiée sous la forme : ***A[] not deadlock***

2- Quand une attaque de déni de service ou une modification de code permanente est détectée, on régénère l'AM :

A[](AS.AS_DoSDetecte or AS.AS_AgentModifie imly A<>AS.AS_AM_Recree)

Dans UPPAAL, nous la notons sous la forme suivante:

(AS.AS_DoSDetecte or AS.AS_AgentModifie) --> AS.AS_AM_Recree

3- Les deux agents coopérants ne sont jamais actifs en même temps :

Il est toujours vrai que les deux agents sédentaires (*AS* et *ASR*) ne coopèrent jamais en même temps avec l'*AM* :

A[] (not (AS.AS_AttenteArrive and ASR.ASR_AttenteArrive))

- Propriétés d'intégrité et de cohérence

Ce sont les propriétés qui ont trait à la cohérence et à l'intégrité du protocole. Autrement dit, le protocole ne pourrait jamais induire en erreur. Par exemple, il ne peut pas détecter une attaque sans qu'il y en ait pas eue. Dans cette catégorie, nous n'avons pas pu exprimer le passé (le temps passé) dans UPPAAL. C'est pourquoi nous avons utilisé des variables mémoires à savoir *attaque* et *panneAS* qui nous ont permis de mémoriser le fait que, dans le temps qui précède un état, une propriété était vérifiée (c'est l'équivalent de *Since* de CTL). Nous citons alors les deux propriétés suivantes :

1- S'il y a détection d'attaque, c'est qu'une attaque a été perpétrée :

La propriété se lit ainsi : il est toujours vrai que si une attaque (ré-exécution de code, un déni de service ou une modification de code simple ou permanente) est détectée, alors cela signifie qu'une attaque est survenue :

A[]((AS.AS_RCDetectee or AS.AS_RCDetectee2 or AS.AS_DoSDetecte or AS.AS_ResIncorrect or AS.AS_AgentModifie or ASR.ASR_RCDetectee or ASR.ASR_RCDetectee2 or ASR.ASR_DoSDetecte or ASR.ASR_ResIncorrect or ASR.ASR_AgentModifie) imply attaque==1)

2- S'il y a détection de panne, c'est qu'une panne est survenue :

Comme la propriété précédente, il est toujours vrai que si l'*ASR* détecte une panne, alors une panne vient de survenir :

A[](ASR.ASR_PanneDetectee imply panneAS==1)

4.1.5 Synthèse

À l'aide du model-checker UPPAAL, nous avons pu vérifier les propriétés énumérées ci-dessus. Cela nous a permis de nous assurer du bon fonctionnement du modèle conçu. Ainsi, toutes les propriétés attendues du système ont pu être validées et, de là, le comportement escompté du protocole. Par ailleurs, le modèle de la plate-forme attaquante reste un modèle très ouvert du fait de sa conception. Alors, nous pouvons lui ajouter toutes les attaques que nous voulons tester à condition de bien concevoir l'attaque en question sous un format adéquat du modèle, autrement dit, en la ramenant à un comportement qui pourrait modifier des variables ou horloges ou qui pourrait éventuellement passer des paramètres à l'un des automates du réseau temporisé.

4.2 Implémentation du protocole

La vérification formelle du protocole étant faite, nous allons à présent faire une évaluation de performance de notre protocole. Pour ce faire, nous allons concevoir les différentes entités constituant le protocole ainsi que leurs comportements. Nous avons effectué notre implémentation en utilisant la plate-forme d'agents mobiles Grasshopper, le logiciel Jbuilder4 pour l'élaboration des classes et des interfaces et le fournisseur de fonctions cryptographiques et des signatures numérique JCE-IAIK. Nous avons donc testé notre protocole en ce qui concerne la détection et la protection contre les attaques précédemment citées, en plus de tester les mécanismes de robustesse et de tolérance aux pannes. Dans cette section, nous allons présenter, dans un premier temps, la plate-forme de test. Dans un deuxième temps, nous allons présenter le modèle conceptuel de notre protocole. Nous enchaînons par la suite avec le plan de test que nous avons adopté et enfin nous présentons les résultats ainsi obtenus, ce qui nous permettra de discuter des performances de notre protocole.

4.2.1 Environnement et choix d'implémentation

Pour implémenter notre protocole, nous avons utilisé la plate-forme d'agents mobiles Grasshopper [IKV98, BAU99]. Elle a été développée en Allemagne par la compagnie IKV++ GmbH en collaboration avec l'institut des systèmes de communication ouverts GMD FOKUS[1]. C'est une plate-forme de développement d'agents mobiles en mode exécution qui est conçue au dessus d'un environnement distribué. Elle est développée en langage Java (basé sur la spécification Java 2). De plus, elle est conçue conformément aux deux standards industriels d'agents mobiles MASIF (pour Management Group's Mobile AgentSystem Interoperability Facility) [OMG95] et FIPA (Foundation for Intelligent Physical Agents) [FIP96].

Dans le cas de notre implémentation, nous avons choisi d'utiliser les sockets pour les interactions entre agents. En plus qu'il est facile à mettre en œuvre, ce choix est motivé par le fait que, pour pouvoir nous comparer au protocole de base, nous devons le simuler dans un environnement d'implémentation similaire avec des options de communication et des choix de protocoles similaires. Comme le protocole de base utilise des sockets, nous avons aussi choisi ce protocole.

Les agents interagissent entre eux par invocation distante. Pour ce faire, nous avons conçu des interfaces via lesquelles, les agents peuvent communiquer. Autrement dit, envoyer un message à un agent consiste en fait à invoquer une méthode de l'agent qui implémente ce message. En effet, chaque agent possède une interface qui contient les méthodes auxquelles il peut réagir. Cependant, avant de commencer la communication, l'agent qui initialise l'interaction devrait d'abord localiser son entité cible. Pour ce faire, il devrait construire un *proxy* de cet objet en le cherchant dans la région. Et de là, il n'aura qu'invoquer la méthode correspondante du message au niveau du *proxy* pour entrer en communication avec l'agent cible.

La Figure 4.1 illustre les différentes étapes pour échanger les messages entre les trois agents *AS*, *AM* et *ASR*. À titre d'exemple, nous présentons une séquence d'échange

[1] http://www.fokus.fhg.de

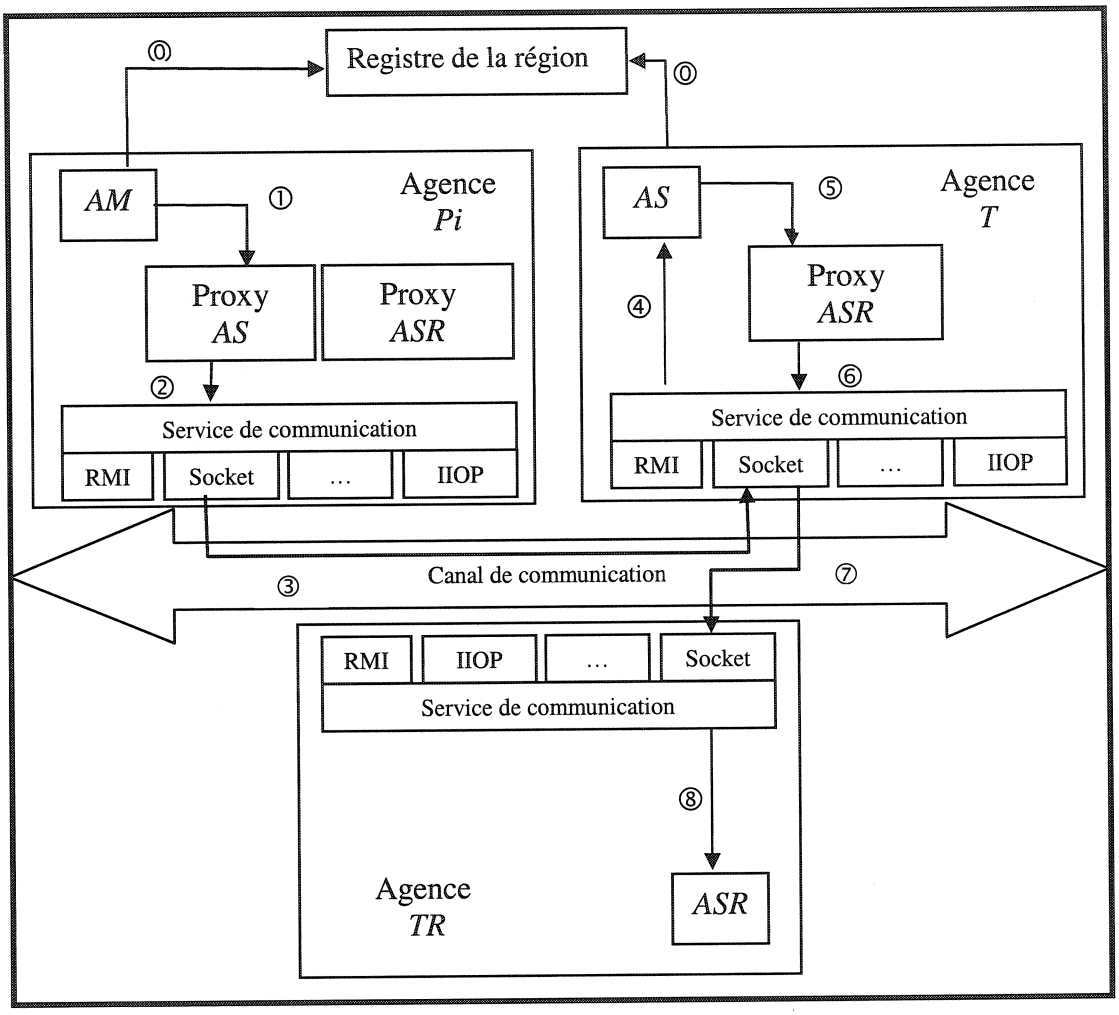

Figure 4.1 Communication entre acteurs du protocole

lors du premier saut de l'agent mobile : au début, les agents recherchent immédiatement les *proxys* de leurs correspondants (étape 0). Ensuite, l'*AM* invoque la méthode qui implémente le message *Arrive()* au niveau de l'objet *proxyAS* (étape 1). Puis, la communication est établie à travers le protocole *Socket* du service de communication au niveau de l'agence P_1 (étape 2) et, en empruntant le canal de communication (étape 3), le message atteint l'objet cible réel *AS* sur la plate-forme *T* (étapes 4). Pour mettre à jour l'*ASR*, l'*AS* invoque la méthode correspondante au message *majASR()* au niveau de

proxyASR (étape 5). Il utilise le service de communication pour acheminer la requête à travers le protocole *Socket* (étape 6) et, en traversant le canal de communication (étape 7), il arrive à l'objet *AS* au niveau de l'agence *TR* (étape 8).

Nous signalons que, dans Grasshopper, la communication peut être soit synchrone soit asynchrone. Dans le premier cas, dit aussi bloquant, l'émetteur doit attendre la réponse du récepteur avant de procéder à autres choses. Par contre, dans le deuxième cas, l'émetteur procède indépendamment des réponses qui pourraient être reçues de son interlocuteur. L'utilisation de l'une ou l'autre des modes de communication dépend étroitement des applications. Dans notre cas, nous avons choisi un mode synchrone, du fait que nous avons trois acteurs qui interagissent et qu'à chaque étape, le déroulement de l'exécution dépend pleinement de l'étape précédente. En fait, le comportement du protocole est réactif aux résultats de chaque étape.

4.2.2 Implémentation du protocole

Après avoir décrit l'environnement d'implémentation, nous avons allons maintenant décrire l'implémentation de chaque composant intervenant dans le protocole. Notre protocole est composé d'agents qui interagissent à travers des interfaces. Ils s'exécutent dans des places (à l'intérieur des agences) fournissant des services de chiffrement et de signatures numériques. Dans cette section, nous allons présenter l'implémentation des places et leurs services de signatures numériques et de chiffrement, puis nous décrirons l'implémentation des agents et leurs interfaces.

4.2.2.1 Implémentation des places

Notre protocole implique la mise en place de quatre types de place selon le rôle assigné. En effet, nous disposons des plates-formes suivantes : une plate-forme d'origine (*PO*), une plate-forme de confiance (*T*), une plate-forme de relève (*TR*), une plate-forme visitée par l'*AM* (P_i) et une plate-forme alternative (P_a) en cas de déni de service de l'*AM*. Sur chacune de ces plates-formes, une agence est nécessaire pour l'accueil des agents. Aussi, dans chaque agence nous avons conçu des places spéciales. Ces dernières

sont dotées de fonctionnalités qui ne sont pas fournies par les places habituelles de Grasshopper. Nous avons ainsi mis en place les places suivantes : *placeCrytoPO* sur *PO*, *placeCryptoT* sur *T*, *placeCryptoTR* sur *TR* et *placeCrypto* sur P_i et P_a.

En ce qui concerne les librairies de sécurité, nous avons utilisé celles offertes par le fournisseur *IAIK_JCE3.03* [IAI02]. C'est un ensemble de librairies qui offrent des API et des implémentations de fonctionnalités cryptographiques. Ces dernières comportent des fonctions de signatures numériques, de chiffrements symétrique et asymétrique ainsi que la gestion et la distribution de clés. Dans le cadre de notre travail, nous avons utilisé l'algorithme de création et de vérification de signature *md5withRSA*. Nous avons également utilisé l'algorithme de chiffrement asymétrique *RSA* [RIV78]. À l'aide de l'outil *keytool* de Sun, nous avons généré des clés privées et publiques de taille 1024 bits pour chaque agence. Pour la gestion et la distribution des clés et des signatures, *keytool* utilise un système local appelé *keystore* qui est en fait une classe java qui maintient des clés pour un système de chiffrement asymétrique. Ce magasin de clés est organisé sous forme de liste indexée par les alias des entités détentrices des clés. Dans notre cas, ces entités ne sont que les places que nous avons citées précédemment.

Ces places utilisent une classe fournie par JDK appelée *KeyStoreHandler*, et c'est elle qui permet la manipulation du *keystore* pour la sauvegarde et l'extraction des clés privées et publiques des agences des plates-formes P_i (pkP_i,$pubkP_i$), *T* (*pkT*,*pubkT*), *TR* (*pkTR*,*pubkTR*), *PO* (*pkPO*,*pubkPO*) et P_a (pkP_a,$pubkP_a$). Aussi, et à l'aide de leur interface commune (*IplaceCrypto*), elles fournissent des services de chiffrement, de déchiffrement, de signature et de vérification aux agents *AS*, *ASR* et *AM*. Leur interface commune est une conséquence directe du fait que les trois agents seront appelés à envoyer et recevoir des messages ; donc ils devront être en mesure d'effectuer les mêmes opérations. La Figure 4.2 présente le diagramme de classes des places qui fournissent lesdits services. Nous notons que la classe *Message* [ELR02] a comme membre un objet de type *SignedObject* de JDK nous permettant ainsi d'effectuer et de vérifier des signatures numériques des messages échangés entre les agents.

4.2.2.2 Implémentation des agents

Nous avons indiqué auparavant que notre protocole est composé de trois agents à savoir l'*AM*, l'*AS* et l'*ASR*. Tous les trois héritent de la classe *MobileAgent* de Grasshopper du fait qu'ils possèdent l'aptitude de mobilité. Chacun d'eux implémente une interface par laquelle une autre entité peut interagir avec lui. Dans ce qui suit, nous allons décrire leurs structures d'implémentation.

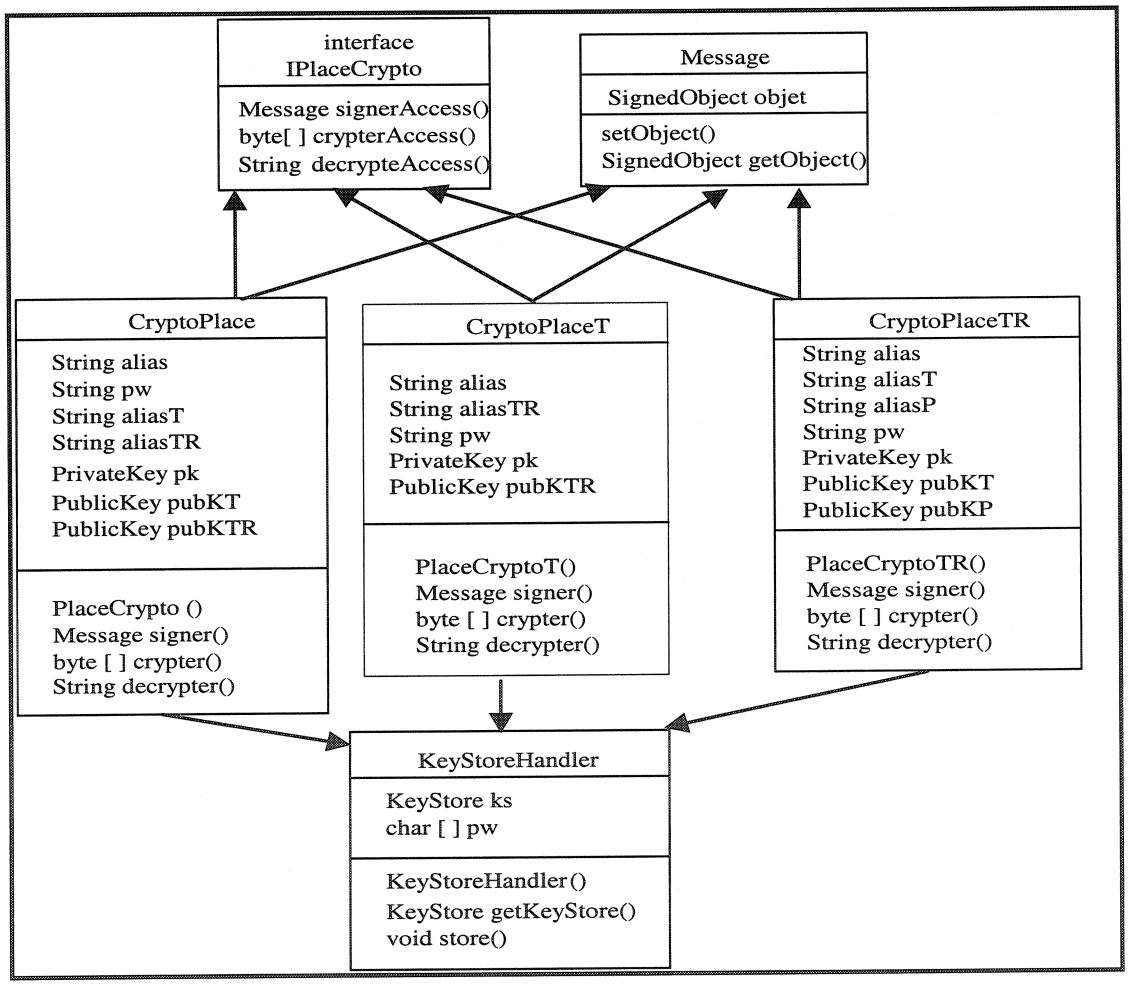

Figure 4.2 Diagramme de classes des places

L'agent AM

La particularité de l'*AM* que nous avons implémenté, c'est qu'il peut être généré dynamiquement par un autre agent et non seulement (comme cela se fait d'habitude) par une plate-forme. Pour ce faire, nous avons modifié la méthode *init()* de la classe mère *MobileAgent* afin de pouvoir lui passer des arguments dynamiquement. Ainsi, un autre agent tel que l'*AS* ou l'*ASR,* quand le cas se présente, pourrait régénérer l'*AM* dynamiquement en lui passant en argument des paramètres comme l'état d'exécution, une destination et/ou d'autres paramètres qui lui seront nécessaires éventuellement pour reprendre une exécution qui n'a pas été achevée par une instance défunte ou encore répondre aux exigences du protocole ainsi conçu. En fait, nous avons exploité pleinement les possibilités qu'offre cette méthode pour l'initialisation de l'*AM*.

En plus des méthodes utilisées dans le protocole de base pour (*crypter(), signer() et vérifier()*), nous avons doté l'*AM* d'une fonction de déchiffrement de messages comme celle des agents *AS* et *ASR*. Ainsi, il pourra déchiffrer d'éventuels messages en provenance de l'*AS* (*killMe()*) ou l'*ASR* (*killMe()* et *changerCooperant()*). Aussi, nous avons développé deux méthodes dans l'interface de l'*AM*, à savoir *changerCooperant()* qui sert à l'*ASR* d'avertir l'*AM* de la panne de l'*AS* pour que tous les messages de coopération, envoyés autrefois vers l'*AS,* soient désormais aiguillés vers lui; et la méthode *killMe()* disponible dans l'interface *I_AM* de l'*AM* sert à l'agent coopérant actif de pouvoir mettre fin à l'*AM* en cas de modification permanente de code. Par ailleurs, la variable *cheminement* permet de gérer l'aspect dynamique de régénération de l'*AM* pour pouvoir distinguer une exécution en cas normal de celle dans des circonstances d'attaque ou de panne. D'autre part, le champ *coopActif* indique à l'*AM* son coopérant qui est actif en cours.

En plus de l'interface de la place cryptographique de l'agence de *Pi, l'AM* utilise la classe *Message* pour tous ses envois et réceptions. Aussi, il utilise les interfaces *I_AS* et *I_ASR* pour interagir avec ses agents coopérants et implémente l'interface *I_AM* pour pouvoir recevoir des messages *(killMe()* et *changerCooperant()).* La Figure 4.3 présente le diagramme de classe de l'agent mobile.

L'agent AS

Tout comme l'agent mobile, l'*AS* utilise aussi la classe *Message* et l'interface *IPlaceCryptoT*. Ainsi, il pourrait, d'une part, utiliser des objets signés dans ses messages et, d'autre part, bénéficier des services cryptographiques, et signatures et de vérification de la plate-forme *T* fournis par la place *PlaceCrtpyoT*. Par ailleurs, pour permettre aux autres agents de lui envoyer leurs messages, l'*AS* implémente des méthodes correspondantes à celles de l'interface *I_AS* qui représentent les messages envoyés par

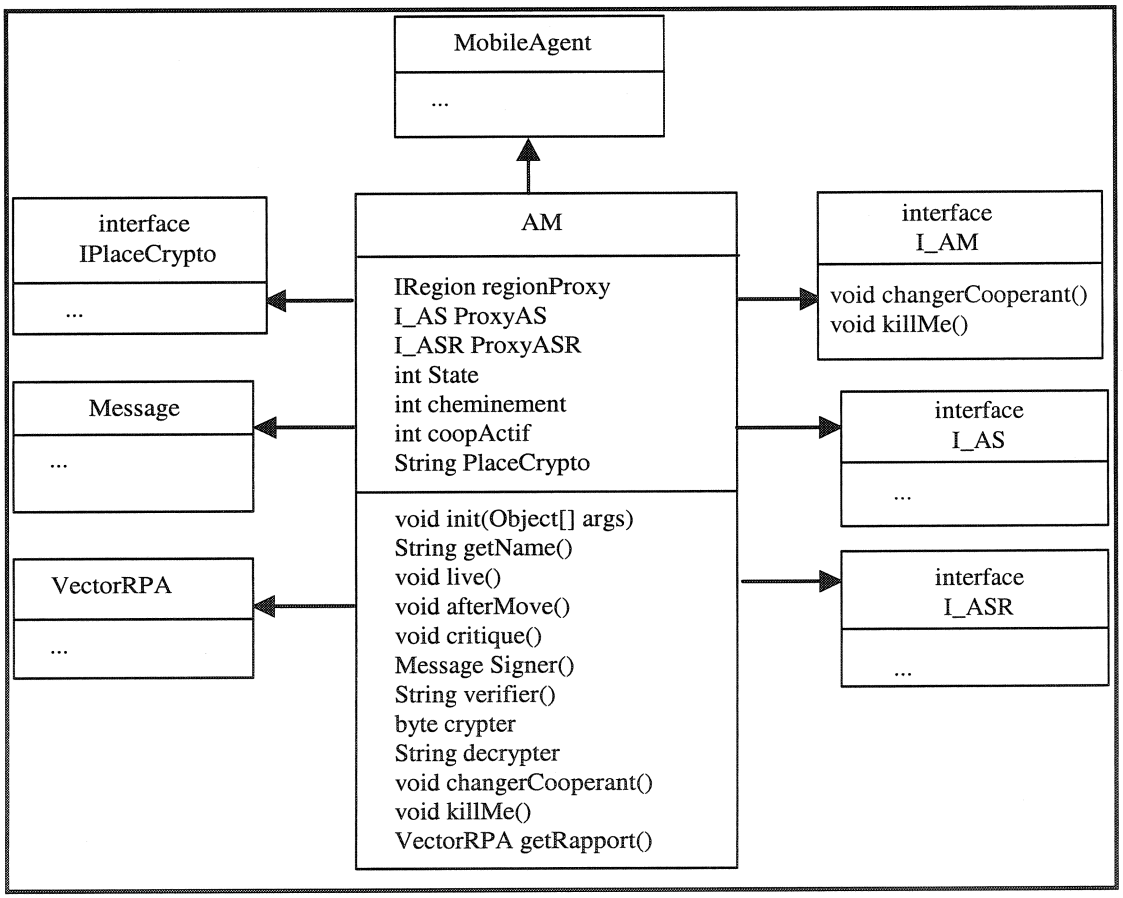

Figure 4.3 Diagramme de classes de l'*AM*

l'agent mobile à savoir *Arrive()*, *Entree()* et *Sortie()*. Aussi, l'*AS* implémente la méthode *critique()* qui exécute le code critique similaire à celui de l'*AM*. Ainsi, il

pourrait comparer les résultats obtenus suite à la réception des entrées nécessaires recueillis par l'agent mobile *AM*. Pour pouvoir générer l'*AM* en présence de circonstances particulières discutées au chapitre précédent, l'*AS* utilise la méthode *genererAM()* qui se charge d'initialiser les arguments (*ArgAM[]*) à fournir à la méthode *init()* de l'*AM*. À la réception du message *Arrive()*, l'*AS* met à jour le *Timeout* qui sert a détecter l'attaque de ré-exécution de code. Pour ce faire, il fait appel à la classe *Timer* [ELR02], qui lui permet d'implémenter un temporisateur qui sera doté de la valeur de *Timeout* mise à jour à la réception du message *Arrive()*. Et plus tard, il lui permet aussi de détecter le déni de service par la valeur de l'*IAS* (intervalle d'attente supplémentaire) [ELR02]. Par ailleurs, l'*AS* envoie, par le biais du message *majASR()*, la liste des attaques perpétrées par la plate-forme en cours (*codeAttaques*) ainsi que l'état de l'*AM* pour qu'en cas de panne de l'*AS*, l'*ASR* saura prendre la relève en bonne et due forme. L'*AS* utilise aussi une variable mémoire (*modifPermanente*) qui lui sert à mémoriser qu'une modification de code est survenue pour pouvoir détecter lors du saut suivant une éventuelle modification permanente de code. En fin de séjour de l'*AM*, ce dernier migre vers la plate-forme *O* et rapatrie le rapport final des résultats et des plates-formes attaquantes auprès de l'*AS* en faisant appel à la méthode *getRaport()* qui utilise la classe *VectorRPA* [ELR02]. Par conséquent, l'*AS* envoie un message *finASR()* pour mettre fin à l'*ASR*. La Figure 4.4 présente le diagramme de classe de l'*AS*.

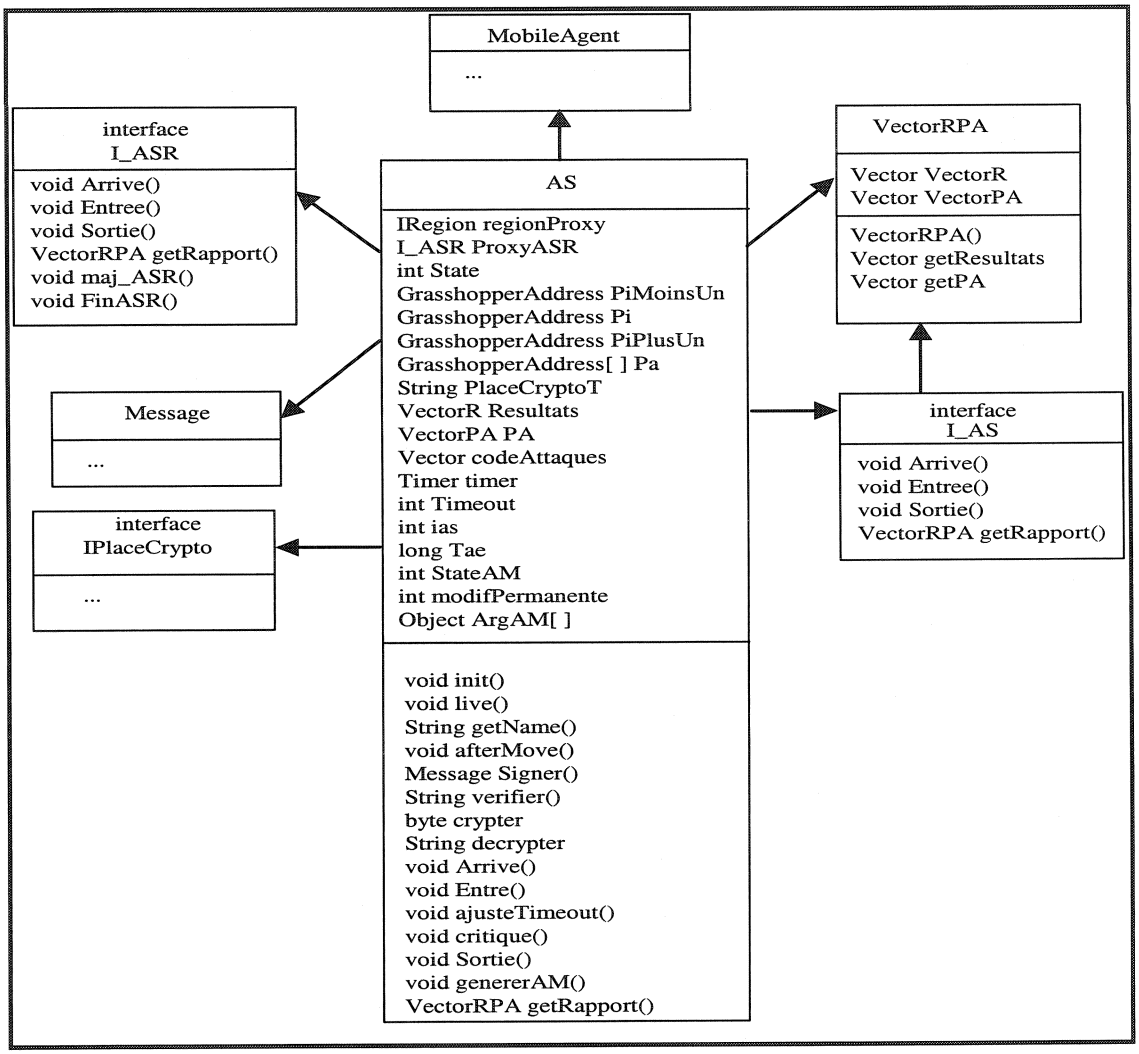

Figure 4.4 Diagramme de classes de l'*AS*

L'agent ASR

L'*ASR* est similaire à l'*AS* sauf pour l'envoi des messages *majASR()*. De plus, l'*ASR* guette la panne de la plate-forme *T* pour prendre la relève. Pour cela, il utilise un *Timer* comme celui de l'*AS*, qui est initialisé à la valeur *TimeoutAS*. Alors, en cas d'expiration du *TimeoutAS*, il invoque une méthode *changerCoopérant()* de l'*AM* pour le mettre au courant de la panne de l'*AS* et pour qu'il change de destinataire de ses

messages. Les variables *Pi*, *PiMoinsUn* et *PiPlusUn* servent à l'*AS* et l'*ASR* pour enregistrer l'itinéraire de l'*AM*. La Figure 4.5 présente le diagramme de classe de l'*ASR*.

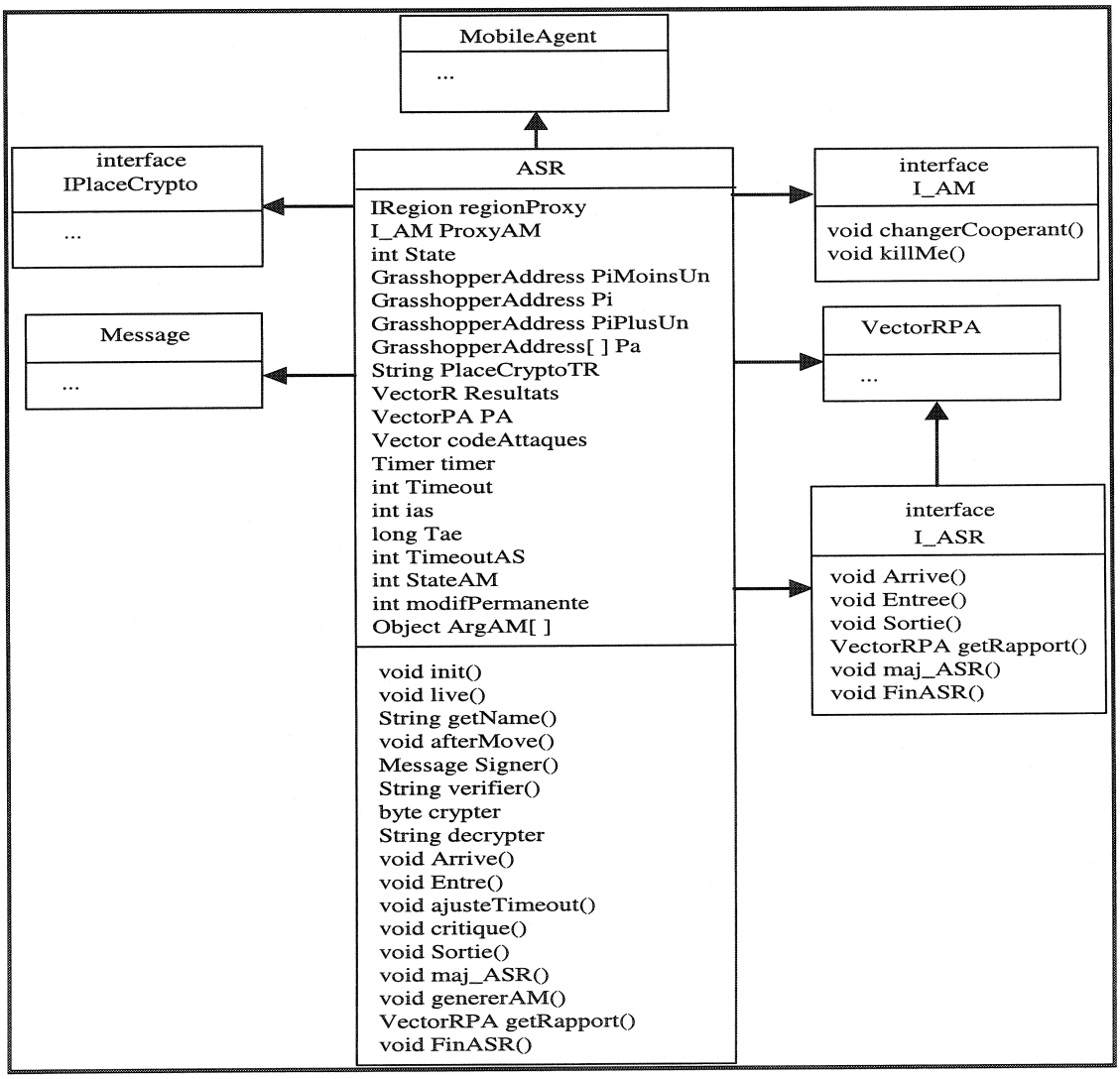

Figure 4.5 Diagramme de classes de l'*ASR*

4.3 Test et évaluation de l'implémentation

Après avoir décrit l'implémentation des entités de notre protocole, nous avons procédé d'abord aux tests préliminaires pour l'estimation du *Timeout* et *TimeoutAS*. Par

la suite, nous avons porté ces résultats aux acteurs du protocole. Enfin, nous avons effectué les tests d'évaluation de performance de l'implémentation de notre protocole.

Nous avons effectué nos tests dans un réseau Ethernet de 100 Mbps. Nous avons utilisé cinq machines identiques dont les caractéristiques sont au Tableau 4.1. Sur chaque machine, nous avons déployé une plate-forme qui consiste en fait en une agence sur laquelle nous avons exécuté la place cryptographique correspondante. Pour la 5ème machine, nous avons fait loger conjointement la plate-forme fiable de reprise *TR* et la plate-forme alternative P_a. Ceci n'a pas d'influence sur le déroulement du protocole vu que c'est seulement un choix secondaire, car nous aurions pu les séparer sans que cela porte atteinte aux performances du protocole.

Tableau 4.1 Emplacement des agences et caractéristiques des machines de test

Type De machine	Nom de la plate-forme	Nom de l'agence/Place	RAM (Mo)	Processeur	Système d'exploitation
Ordinateur personnel	Plate-forme d'origine *O*	*Agence0/ CryptoPO*	512	P IV 1.6 GHz	Windows 2000 pro
Ordinateur personnel	Plate-forme de confiance *T*	*Agence / PlaceCryptoT*			
Ordinateur personnel	Plate-forme P_1	*Agence1 / PlaceCrypto*			
Ordinateur personnel	Plate-forme P_2	*Agence2 / PlaceCrypto*			
Ordinateur personnel	Plate-forme alternative P_a	*Agence3 / PlaceCrypto*			
	Plate-forme de confiance de reprise *TR*	*AgenceR / PlaceCryptoTR*			
--------+----------+--(Réseau Ethernet 100Mbps)--+----------+----------					

4.3.1 Scénarios de test

Nous rappelons qu'au chapitre précédent nous avons conçu des places cryptographiques qui fournissent des fonctions de sécurisation. Celles-ci sont des parties intégrantes du protocole. Alors, quand l'*AM* s'exécute sur une plate-forme P_i et avant

d'envoyer son premier message *Arrive()*, il doit d'abord activer le contexte de sécurisation, i.e. le fournisseur *IAIK_JCE3.03* [IAI02] qui fournit les éléments que manipulent les fonctions de sécurisation. Par conséquent, sur toutes les plates-formes que l'*AM* va visiter, il faudrait donc que l'*AM* active ce contexte avant d'entamer son exécution : c'est ce que nous appelons l'aspect «cache ».

Par ailleurs, nous appelons « proxy » l'image de l'agent coopérant que l'*AM* doit construire pour localiser son coopérant avant de lui envoyer les messages de coopération. En fait, la construction du proxy, consiste à rechercher dans toute la région l'interface *I_AS* qu'implémente l'*AS*. Ainsi, l'*AM* pourrait invoquer la méthode *arrive()* de *I_AS* implémentant le message *Arrive()*. Ceci est vrai pour le premier saut car, une fois dans la plate-forme P_{i+1}, l'*AM* possède déjà l'adresse du proxy, donc il n'a plus besoin de le chercher. Aussi, lors de la reprise après un déni de service, l'*AM* est obligé de construire le proxy de l'agent coopérant avant de procéder à la coopération.

Par conséquent, nous avons pris en compte lors de nos tests sont reliées à ces deux aspects, à savoir : le proxy des correspondants d'un agent et l'activation des fonctions de sécurisation des places cryptographiques (cache). Ainsi, nous retenons les deux scénarios de test suivants :

- sans cache et sans proxy (pire cas) ;
- sans cache et avec proxy (meilleur cas).

4.3.2 Estimation dynamique de la valeur du *Timeout*

À la réception du message *Arrive()*, l'*AS* procède au premier relevé en vue de l'estimation de la valeur du *Timeout*. Pour ce faire, il marque le temps d'arrivée du message *Arrive()*. Quand il reçoit le message *Entree()* en provenance de l'*AM*, il relève ce temps d'arrivée. Il calcule ensuite le temps inter-arrivée T_{ae} des deux premiers messages (*Arrive()* et *Entree()*). À la réception du message *Sortie()*, il marque le temps d'arrivée et en déduit le temps d'inter-arrivée T_{es} des deux derniers messages (*Entree()* et *Sortie()*). À la fin de l'exécution de l'*AM*, nous relevons le temps d'exécution total *Temps* sur la plate-forme en question. Ainsi, nous avons les quatre valeurs suivantes :

T_{ae}, T_{es}, (T_{ae}+ T_{es}) et *Temps*. Ensuite, nous avons entrepris une série de mesures dans les deux scénarios considérés pour essayer de trouver une loi empirique qui pourrait relier d'une part T_{ae} et T_{es} et d'autre part (T_{ae}+ T_{es}) et *Temps*. Ces fonctions nous les noterons :

$$T_{es} = f(T_{ae}) \text{ et } Temps = g(T_{ae} + T_{es}).$$

Cas sans cache et avec proxy (meilleur cas)

Dans un premier temps, les résultats que nous avons obtenus par rapport aux mesures de $T_{es} = f(T_{ae})$ montrent bien une corrélation entre T_{ae} et T_{es}. La Figure 4.6 illustre cette relation. Nous avions pu approcher le nuage de points obtenu selon une approximation linéaire dont l'équation est la suivante :

$$\mathbf{T_{es} = -0.2079 * T_{ae} + 307.53} \qquad (4.1)$$

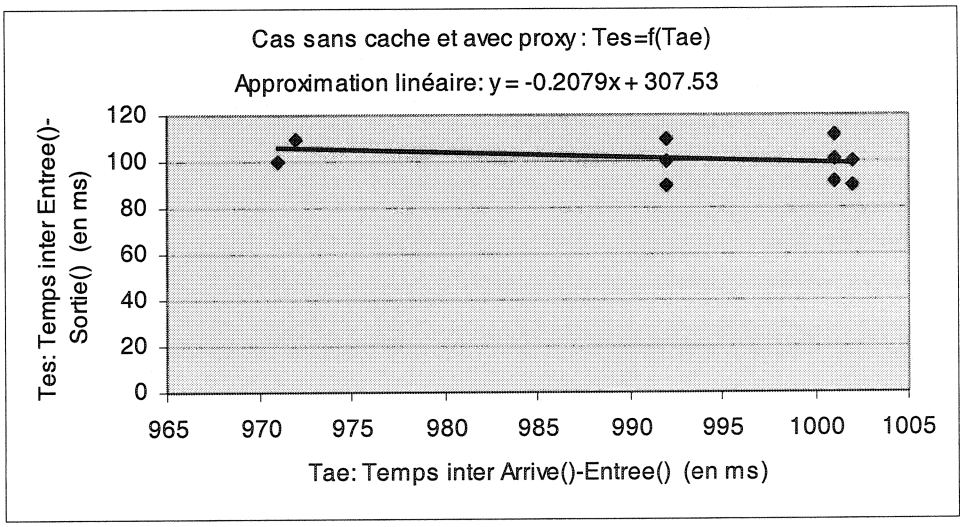

Figure 4.6 Corrélation $T_{es} = f(T_{ae})$

Nous constatons que les valeurs ne sont pas trop dispersées. Il y a un écart d'environ 20 ms et 30 ms respectivement dans la direction de T_{ae} et T_{es}. Aussi, nous remarquons que le temps T_{ae} est supérieur à T_{es}. Cela est dû au fait qu'au début de l'exécution, c'est la méthode *afterMove()* de l'*AM* qui est exécutée et c'est à ce niveau

que l'*AM* envoie le message *Arrive()*. Ensuite, c'est la méthode *live()* qui s'exécute après et c'est là que les deux messages *Entrees()* et *Sortie()* sont invoqués, ce qui explique cet écart. En effet, les deux derniers messages sont invoqués depuis la même méthode *live()*.

Dans un deuxième temps, nous avons obtenu les résultats qui sont illustrés à la Figure 4.7 et qui représente la relation *Temps* = g (T_{ae}+ T_{es}). Nous remarquons que le paramètre *Temps* est proportionnel à la somme T_{ae}+ T_{es}, une chose qui est explicable du fait que, si les messages échangés (représentée par (T_{ae}+ T_{es})) prennent du temps pour arriver à destination, c'est que l'exécution de l'*AM* (*Temps*) a dû prendre du temps.

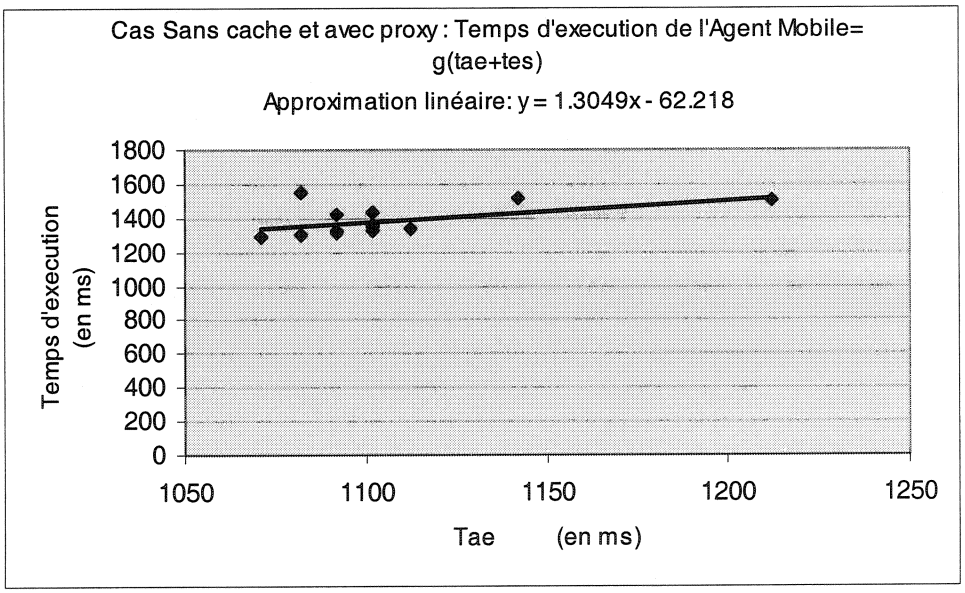

Figure 4.7 Corrélation *Temps* = g (T_{ae}+ T_{es})

L'écart est dû en grande partie au fait que le temps nécessaire à l'*AM* pour construire le proxy ainsi que le temps de démarrage du service de la place *PlaceCrypto* de la plate-forme P_i n'est pas comptabilisé dans la valeur de (T_{ae}+ T_{es}). En fait, la prise de relevé de cette dernière (T_{ae}+ T_{es}) ne commence qu'après l'arrivée du message *Arrive()*, ce qui fait que les procédures de recherche de proxy et de démarrage des

services cryptographiques de la place ne sont pas prises en compte dans la mesure. Par contre, dans le cas de l'autre paramètre (*Temps*), toutes les étapes sont comptabilisées.

Nous obtenons alors, avec une approximation linéaire, la relation qui lie les deux paramètres. Elle est donnée par la relation suivante :

$$\boldsymbol{Temps = 1.3049*(T_{ae} + T_{es}) - 62.218} \qquad (4.2)$$

Ainsi, en combinant les deux relations (4.1) et (4.2), nous obtenons la relation qui relie T_{ae} à *Temps* comme suit :

$$\boldsymbol{Temps = 1.033*T_{ae} + 401.29} \qquad (4.3)$$

En fait, nous avons estimé le temps d'exécution de l'*AM* sur une plate-forme P_i dans le cas où l'*AM* a déjà le proxy de l'agent coopérant. L'estimation est faite en fonction du temps d'inter-arrivée des deux premiers messages (*Arrive()* et *Entree()*). Afin d'estimer une valeur de notre *Timeout*, nous avons procédé à une série de mesures pour comparer le temps d'exécution effectif de l'*AM* avec le temps estimé selon la relation (4.3). En effet cette dernière a été introduite dans le code de l'agent sédentaire pour présenter le *Timeout* et pour laquelle nous avons effectué des redressements à l'aide de coefficients multiplicateurs. En comparant les résultats obtenus, nous avons calculé l'écart moyen par rapport au temps effectif d'exécution pour aboutir aux résultats de la Figure 4.8. Nous pouvons voir que l'écart moyen de l'estimation corrigée par un coefficient de 0.975 est le meilleur en valeur absolue, ce qui fait que nous choisirons cette solution pour notre protocole dans le cas sans cache et avec proxy, tout en étant assuré qu'elle approchera le mieux le temporisateur que détiendra l'agent sédentaire.

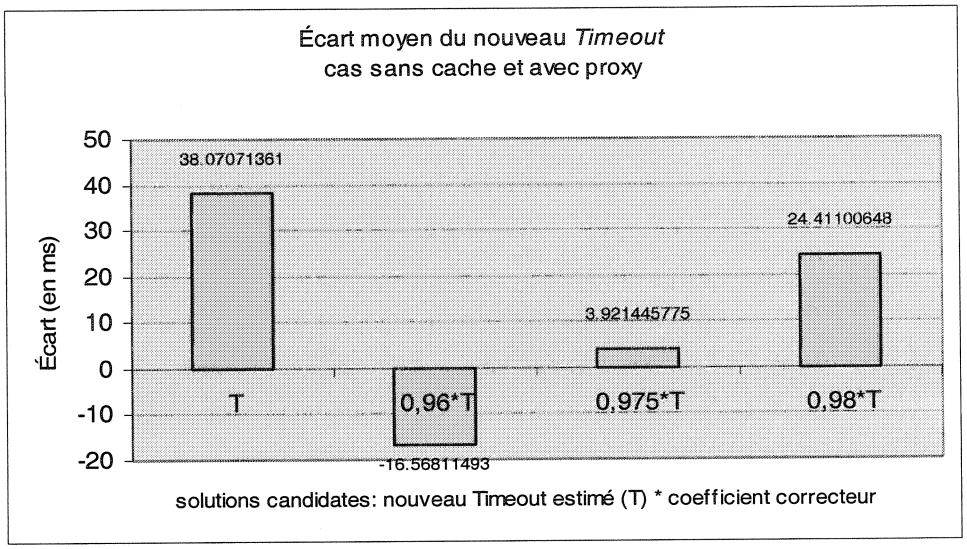

Figure 4.8 Écart moyen entre les différents redressements du Timeout estimé et le temps d'exécution effectif

Pour pouvoir nous comparer au protocole de base, nous avons reproduit la simulation faite dès lors et nous avons effectué les mêmes essais que notre protocole pour obtenir les écarts moyens par rapport au temps d'exécution. Le choix des coefficients multiplicateurs est basé sur l'étude faite lors des travaux du protocole de base qui a conclu que le meilleur intervalle est situé entre 1,3 et 1,5 où le taux de détection de l'attaque est supérieur à 90% et le taux où l'agent coopérant se trompe est égal à 0%. Cependant, pour être plus rationnel nous avons étendu cet intervalle pour obtenir une comparaison à la fois correcte et raisonnable. Alors, nous avons obtenu les résultats de la Figure 4.9. Nous constatons que le meilleur résultat se situe loin à l'extérieur de l'intervalle stipulé par le protocole de base. De plus, le coefficient 0.6 donne bien une idée de la qualité de l'estimation faite du *Timeout* qui réduit l'estimation faite presque de moitié, sans négliger bien sur l'écart obtenu de 14.92 ms très grand par rapport à notre estimation qui donne seulement un écart de 3.92 ms avec un coefficient de 0.975 (très proche de 1) et qui révèle bien la qualité de notre solution à ce stade.

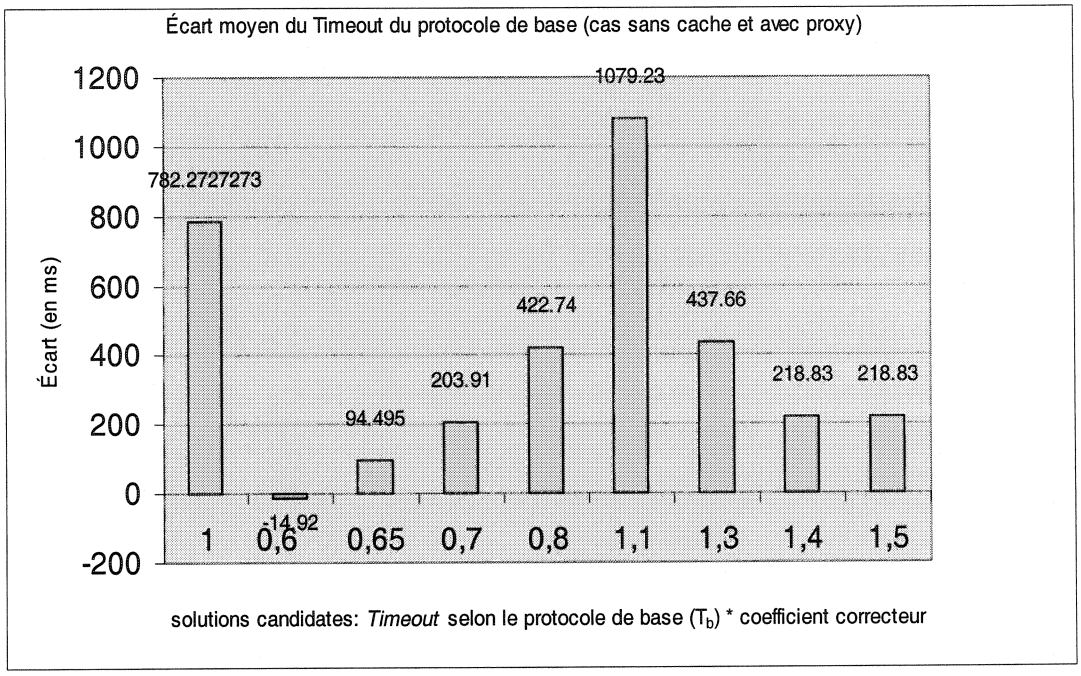

Figure 4.9 Écart moyen des différents redressements de l'estimation du Timeout estimé dans le protocole de base par rapport au temps d'exécution

Cas sans cache et sans proxy (pire cas)

Comme dans le cas précédent, nous avons obtenu les corrélations $T_{es} = f(T_{ae})$ et $Temps = g(T_{ae} + T_{es})$. Pour la première, les résultats sont présentés à la Figure 4.10. À l'aide d'une approximation linéaire, nous avons obtenu la corrélation définie par l'équation suivante :

$$T_{es} = 0.0429 * T_{ae} - 185.65 \qquad (4.4)$$

Il est clair que T_{es} suit linéairement T_{ae}, ce qui est naturel comme tendance. Si la même plate-forme prend plus de temps pour exécuter le code de l'*AM* avant d'envoyer le message *Entree()* (T_{ae}), alors elle mettra un temps proportionnel pour exécuter la deuxième partie du code de l'*AM*, étant donné que les deux codes sont de même niveau de complexité.

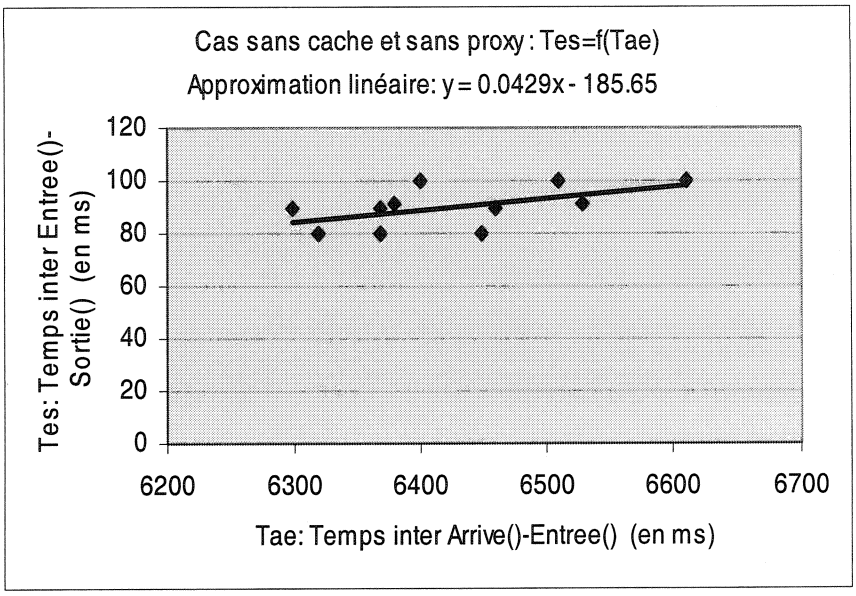

Figure 4.10 Corrélation $T_{es}=f(T_{ae})$ dans le cas sans cache et sans proxy

Pour la deuxième relation qui lie *Temps* et $T_{ae}+T_{es}$, nous avons effectué une série d'expériences qui nous ont permis d'obtenir les résultats de la Figure 4.11. L'approximation linéaire est régie par la relation suivante :

$$Temps = 1.0241*(T_{ae}+T_{es}) + 216.49 \qquad (4.5)$$

Nous remarquons bien la loi linéaire qui relie les deux paramètres. En combinant les deux relations (4.4) et (4.5), nous déduisons la relation qui relie *Temps* à T_{ae} :

$$Temps = 1.068*T_{ae} + 26.36 \qquad (4.6)$$

Ainsi, nous avons pu estimer le temps d'exécution de l'*AM* en fonction du temps d'inter-arrivée des messages *Arrive()* et *Entree()*. Par la suite, nous avons introduit cette

relation dans le code de l'agent coopérant pour pouvoir estimer dynamiquement le *Timeout*.

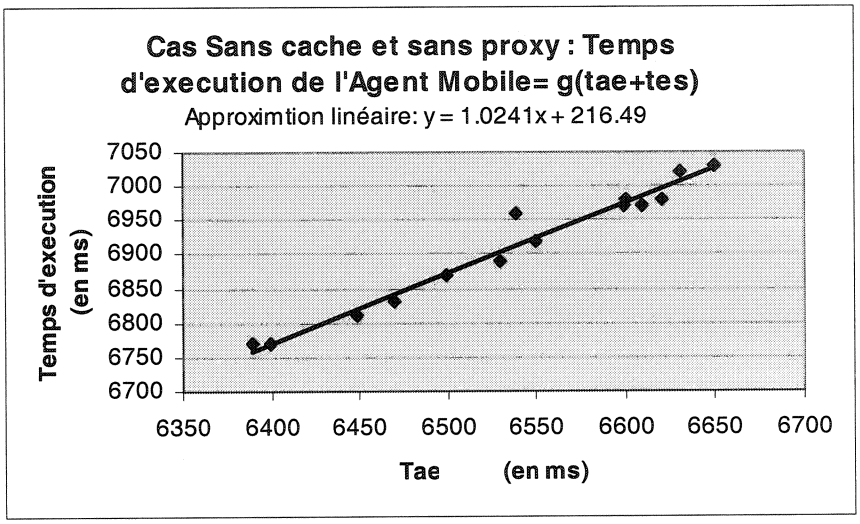

Figure 4.11 Corrélation *Temps* = *g* (T_{ae} + T_{es}) dans le cas sans cache et sans proxy

Finalement, pour pouvoir se comparer au protocole de base dans ce cas, il a fallu voir les écarts par rapport au temps d'exécution effectif et, à l'aide de coefficients correcteurs, nous obtenons l'approximation adéquate du *Timeout* que nous adopterons dans notre protocole. De la même façon que dans le cas sans cache et avec proxy, les coefficients pour le protocole de base seront ceux indiqués dans le cas précédent car ce protocole ne distingue pas ces deux cas de figure. Nous avons obtenu les résultats de la Figure 4.12 où nous avons présenté les deux approches et nous avons porté les valeurs des estimations pour voir l'écart par rapport au temps d'exécution. La qualité de l'approximation est très claire : celle du protocole de base pénalise une plate-forme moins rapide. Pour notre protocole, l'estimation corrigée ne pénalise pas une plate-forme moins performante ni ne laisse plus de chance à une plate-forme rapide puisque l'écart est minime. Ainsi, pour l'évaluation de performance, nous adopterons respectivement les coefficients 1.001 et 2 pour notre protocole et le protocole de base.

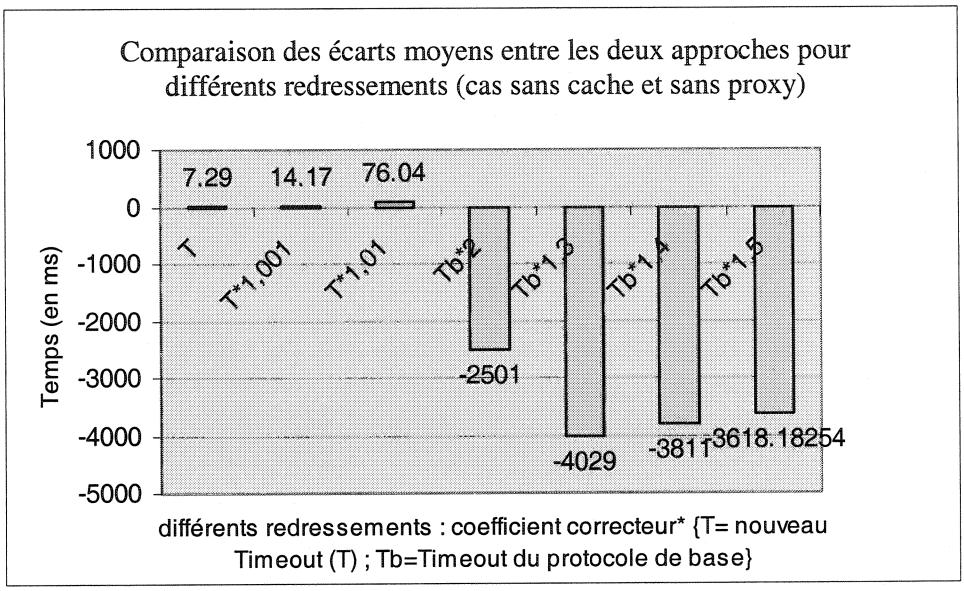

Figure 4.12 Comparaison des écarts moyens, par rapport au temps d'exécution, des estimations du *Timeout* dans les deux protocoles

Pour résumer, nous signalons que le protocole choisit dynamiquement le coefficient à adopter selon le scénario. Ainsi, pour une première exécution et lors d'une exécution après une panne ou un déni de service, le protocole choisit l'estimation du pire cas et, pour toute autre exécution, il adopte l'estimation du meilleur cas.

4.3.3 Estimation de l'*IAS* et *TimeoutAS*

Nous allons présenter maintenant notre approche pour l'estimation de la valeur de l'*IAS*, i.e. le temps d'attente supplémentaire pour que l'agent coopérant puisse déclarer qu'un déni de service est survenu. Nous allons aussi présenter l'estimation de la valeur du *TimeoutAS* au bout duquel l'*ASR* déclarera qu'une panne de la plate-forme *T* est survenue.

Nous signalons que la valeur de *TimeoutAS* dépend de celle de *IAS*, car si l'*AS* détecte un déni de service, il devra envoyer un message *majASR()* dans lequel il indiquera le type d'attaque. Cependant, si l'*ASR* déclare déjà une panne de *T* alors qu'il y avait un déni de service, il sera trompé sur sa décision. Or, la valeur de l'*IAS* ne relève

que du bon sens car cela dépend des cas. Il faut donc laisser un temps raisonnable avant de déclarer un déni de service, par exemple, multiplier la valeur du temporisateur par un coefficient 2 ou peut être 3/2. Ce que nous avons fait c'est estimer le temps d'inter-arrivée des messages *majASR()*. Au début de l'activation du protocole, l'*ASR* migre vers la plate-forme *TR*. Quand l'*AS* est créé dans la plate-forme d'origine, il envoie un message *majASR()*, puis il migre vers la plate-forme *T*. Ensuite, il simule l'exécution de l'*AM* pour envoyer enfin un deuxième message *majASR()*, ceci avant le lancement de l'*AM*.

Nous avons effectué une série d'expériences pour les deux scénarios étudiés afin de corriger ce temps estimé. Nous avons obtenu les résultats des Figures 4.13 et 4.14. Dans le cas sans cache et avec proxy (c.f Figures 4.13), les écarts sont positifs pour les deux coefficients 1.7 et 1.8, ce qui veut dire que le protocole laissera plus de temps que l'estimation faite du temps d'inter-arrivée de deux messages *majASR()*. Les probabilités que le protocole se trompe est de 12.5% dans le cas de 1.7 et de 0% pour 1.8. Ainsi, nous adopterons le coefficient correcteur de 1.8, ce qui nous donnera l'assurance que le temporisateur n'expirera pas avant, suite à un retard ou une ré-exécution de l'*AM*.

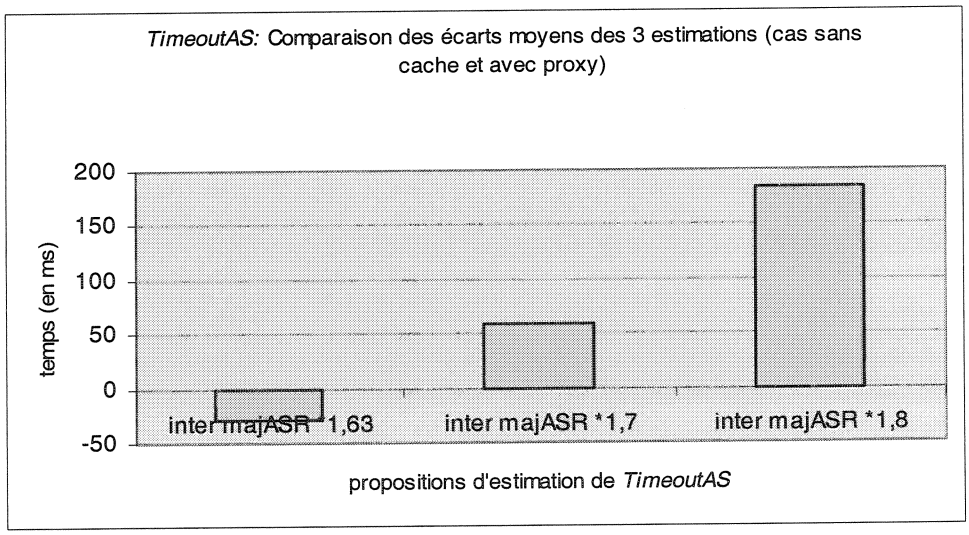

Figure 4.13 Comparaison des écarts moyens des estimation de *TimeoutAS* par rapport au temps d'inter-arrivée des *majASR()* (cas sans cache et avec proxy)

Dans le cas de la Figure 4.14, nous avons présenté ces deux propositions car ce sont celles qui approchent le mieux le temps effectif. Les écarts sont positifs certes, mais la probabilité pour que le protocole se trompe est de 0% pour un coefficient 6.15 alors qu'elle est de 14.28% pour celui de 6.10. Ainsi, nous adoptons la première proposition pour notre protocole.

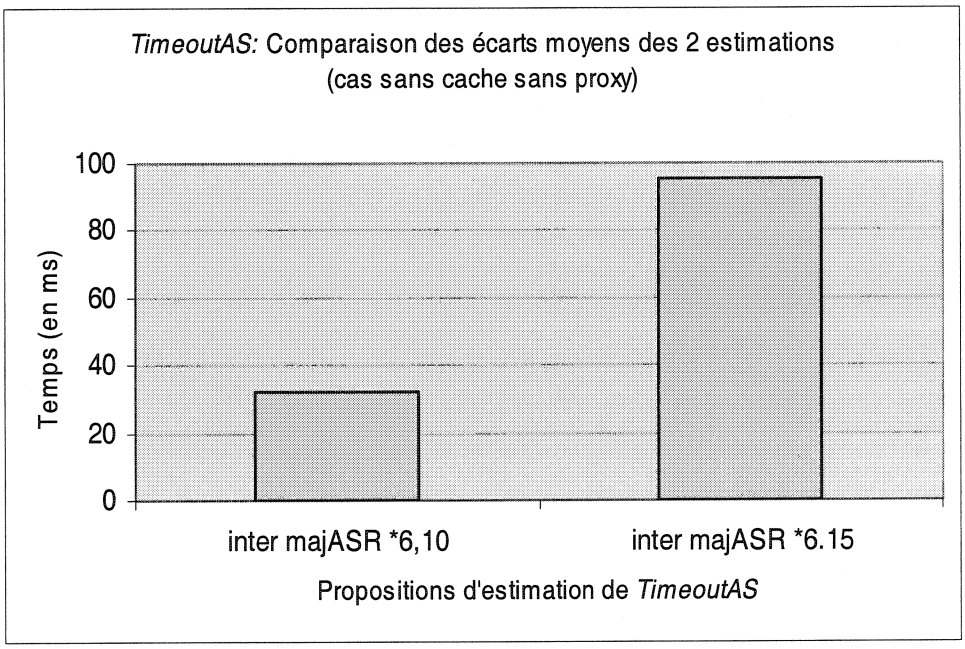

Figure 4.14 Comparaison des écarts moyens des estimation de *TimeoutAS* par rapport au temps d'inter-arrivée des *majASR()* (cas sans cache et sans proxy)

Comme pour l'estimation du *Timeout*, notre protocole choisit un des coefficients élus selon le scénario dans lequel il se trouve.

4.3.4 Test de l'implémentation

Pour tester les mécanismes et les fonctionnalités de notre protocole, nous avons entrepris des expérimentations dans lesquelles nous avons conçu des scénarios de pannes ou d'attaques par des plates-formes malicieuses. Chaque scénario présente une situation

particulière qui permet de tester un comportement ou un mécanisme intégré à notre protocole. La mise en œuvre de ces scénarios, en l'absence de plate-forme malicieuse, a consisté à modifier le code de l'*AM* pour le forcer à simuler un comportement malicieux et nous avons observé le comportement du protocole. Le Tableau 4.2 présente les scénarios d'attaques effectuées ainsi que les résultats obtenus. Seules les attaques de modification avec combinaison de deux ou trois éléments de coopération avec l'*AS* et l'attaque intelligente de changement d'itinéraire (la deuxième) ne sont pas détectées par l'agent coopérant.

Tableau 4.2 Mise en œuvre des attaques et réactions de notre protocole

Type de l'attaque	Scénario de l'attaque	Mise en œuvre de l'attaque	Réaction de notre protocole
Changement d'itinéraire	L'*AM* est envoyé vers une autre plate-forme	*AM* migre vers P_a (au lieu de P_2) avec $Suiv_1=P_2$	Détectée par l'*AS* immédiatement si P_a ne collabore pas avec P_1
	P_1 attaque intelligemment l'*AM*	*AM* migre vers P_a (au lieu de P_2) avec $Suiv_1=P_a$	N'est pas détectée par l'*AS*
Déni de service	P_1 retarde l'*AM*	P_1 exécute la méthode *sleep()* ou exécute une boucle très longue	Détectée par *AS* après expiration de *Timeout* et *IAS*
	P_1 capture l'*AM*	*AM* ne complète pas son exécution et n'envoie pas les messages *Entree()* et *Sortie()*	Détectée par *AS* après expiration de *Timeout* et *IAS*
Mascarade	Mascarade de la plate-forme P_1	*AM* envoie une identité autre que celle de P_1	Détectée par *AS* après la vérification du message *Arrive()*
	Un intrus *AM'* se fait passer pour l'*AM*	*AM'* envoie des messages à *AS*	Détectée par *AS* après la vérification de la signature
Modification	Modifie les entrées	P_1 envoie à *AS* des données différentes de celles que l'*AM* détient	Détectée par l'*AS* à la réception du message *Sortie()* *ResAM<> ResAS*
	Modifie les sorties	P_1 envoie à *AS* des résultats différents de ceux que l'*AM* a obtenus	Détectée par *AS* à la réception du message *Sortie()* *ResAM<>ResAS*
	Modifie seulement le code critique	P_1 envoie à *AS* des entrées correctes et les résultats effectivement obtenus	Détectée par l'*AS* à la réception du message *Sortie()*; *AS* créé un nouvel *AM* et l'envoie vers P_a
	Combinaison de modifications de *Entrée()* et/ou code critique et/ou *Sortie()*	P_1 modifie les trois éléments à la fois (*Entrée()*, code critique et *Sortie()*) ou deux éléments en même temps	L'attaque ne peut être détectée
	Modifie l'agent en permanence après la fin de l'exécution	P_1 exécute *AM*, ensuite elle le modifie. *AM* migre à P_2 qui exécute un code critique modifié, il migre vers P_a qui fait la même chose. Il migre ensuite vers P_1	Détectée par l'*AS* au 3ème saut si P_2 et P_a ne collaborent pas avec P_1 : *AS* reçoit des résultats incorrects de P_a. Il élimine *AM* (*killMe()*) sur P_1 puis crée un *AM* et l'envoie vers P_a et examine les résultats qui seront corrects.
	Modifie l'agent en permanence avant la fin de l'exécution	P_1 modifie *AM* et l'exécute. *AM* migre ensuite à P_2 qui l'exécute sans collaboration avec P_1. *AM* migre ensuite à P_a	Détectée par l'*AS* au 2ème saut si P_2 ne collabore pas avec P_1 : *AS* reçoit des résultats incorrects de P_2. Il élimine *AM* (*killMe()*) sur P_a puis crée un *AM* et l'envoie vers P_2 et examine les résultats qui seront corrects.
Ré-exécution de code	P_1 re-exécute le code critique de l'*AM*	*P1* exécute deux fois la méthode *critique()* de *AM*	Détectée par *AS* après expiration de *Timeout*

Le Tableau 4.3 présente les deux tests de pannes auxquelles le protocole peut réagir. Le mécanisme de survivabilité du protocole les prend en charge complètement.

Tableau 4.3 Mise en œuvre des pannes et réactions de notre protocole

Scénario de la panne	**Mise en œuvre de la panne**	**Réaction de notre protocole**
Panne de la plate-forme *T*	Nous mettons fin à «*Agence*» aussitôt que le premier *majASR()* est envoyé à l'*ASR*	L'*ASR* détecte la panne après expiration de *TimeoutAS*, il prend la relève et envoie un message *changerCooperant()* à *AM*
Panne de l'*AS* sur la plate-forme *T*	AS n'envoie plus de messages *majASR()*	l'*ASR* détecte la panne après expiration de *TimeoutAS*, il prend la relève et envoie un message *changerCooperant()* à *AM*

4.3.5 Évaluation de l'implémentation

Nous avons évalué les performances de notre protocole relativement au protocole de base. Pour ce faire, nous avons considéré les aspects suivants :

- la détection de la ré-exécution de code pour pouvoir mesurer la qualité de l'approximation du *Timeout* ;
- le trafic généré entre les plates-formes qui participent au protocole.

Analyse du raffinement de la detection de la ré-exécution de code

Après les comparaisons entre notre protocole et le protocole de base quant aux qualités de l'estimation du *Timeout*, nous avons jugé utile de faire une autre

comparaison en présence de plate-forme qui ré-exécute le code intentionnellement. Selon les deux scénarios considérés, nous avons obtenu les résultats suivants :

1. Cas sans cache et sans proxy : Nous avons mesuré le taux de détection avec les coefficients de redressement adoptés dans ce cas, à savoir : 1.001 dans le cas de notre protocole et 2 pour le protocole de base. Les deux protocoles ont un taux de détection de 100%. Les taux sont satisfaisants certes; cependant, l'écart du *Timeout* est très grand dans le cas du protocole de base car il faut savoir que la ré-exécution de code de l'agent mobile consiste seulement à la ré-exécution de son code critique (*critique()*) qui est en fait une petite fonction qui ne représente que peu par rapport au temps d'exécution global de l'*AM*. Et de là, dans d'autres circonstances, si l'*AM* s'exécute sur une plate-forme lente, le protocole risquera de se tromper vu que l'estimation du *Timeout* est trop courte et pénalisera ainsi les plates-formes moins performantes.

2. Cas sans cache et avec proxy : avec des coefficients de redressement de 0.975 pour notre protocole et 0.65 pour le protocole de base. Nous avons obtenu des taux de détection de 35.3% pour le protocole de base et de 100% pour notre protocole, ce qui confirme la qualité de notre approche. De plus, notre protocole possède un écart moyen raisonnable par rapport au temps d'exécution de la plate-forme attaquante. Ainsi, il ne pourrait pas se tromper par rapport à l'intégrité d'une plate-forme lente sans laisser passer inaperçues des plates-formes malveillantes très performantes.

Analyse du trafic généré

Pour mesurer le trafic généré entre les différentes plates-formes participantes, nous avons utilisé le logiciel *Ethereal*[3]. Ce dernier se met à l'écoute pour enregistrer tous les paquets échangés qui transitant dans le réseau. Avec les possibilités de filtrage sur les adresses IP et les ports qui entrent en action, nous avions pu identifier chaque trafic à

[3] http://www.ethereal.com/

part et, de là, nous l'avons mesuré. La Figure 4.15 présente le trafic généré durant la durée de vie de l'agent mobile.

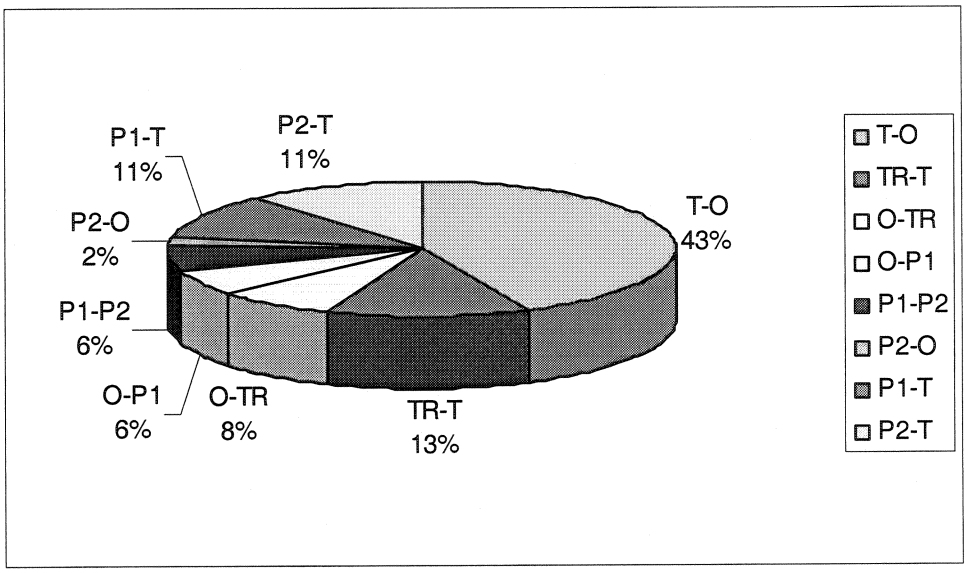

Figure 4.15 Répartition du trafic généré entre les plates-formes

Nous constatons que le trafic est généré en grande partie (43%) des échanges entre la plate-forme d'origine *O et* celle de confiance *T*. Ceci est dû au fait que, dans la plate-forme *T*, deux entités s'exécutent, d'une part l'agence «*Agence*» qui abrite l'agent *AS* et, d'autre part, la région à laquelle appartiennent toutes les agences, places et agents du protocole. Ainsi, les déplacements de tous les agents devront y être reportés (registre de région). Vu que les migrations sont au sens faible (par instanciation et pas de déplacement du code), les agents sont donc instanciés au niveau des plates-formes visitées, alors que le code reste sur la plate-forme d'origine. Cette dernière devra alors envoyer des paquets à *T* pour mettre à jour le registre de région, en particulier l'*AM* qui effectue plusieurs migrations. Par ailleurs, le trafic généré suite aux échanges entre *T* et P_1 est égal en pourcentage (par rapport au trafic total du protocole) à celui entre *T* et P_2. Cela est dû au fait que le rôle que joue P_1 est similaire à celui de P_2. Cependant, le

protocole ajoute un trafic de 13% consistant aux échanges qui découlent de l'ajout de l'*ASR*, trafic pourrait être compensé en partie par l'élimination de l'agent estimateur *AE* du protocole de base.

D'un autre côté, à la Figure 4.16 nous présentons les tailles des agents dans les deux protocoles. Nous constatons que notre protocole présente une augmentation de 11.7% de la taille de l'*AS* par rapport à celui du protocole de base. De plus, en ce qui concerne l'*ASR*, sa taille est de 14 *Ko*, équivalente au double de la taille de l'agent estimateur *AE* du protocole de base dont nous avons éliminé l'utilité. En partie, cela constitue le prix à payer pour avoir un protocole plus sécuritaire et robuste.

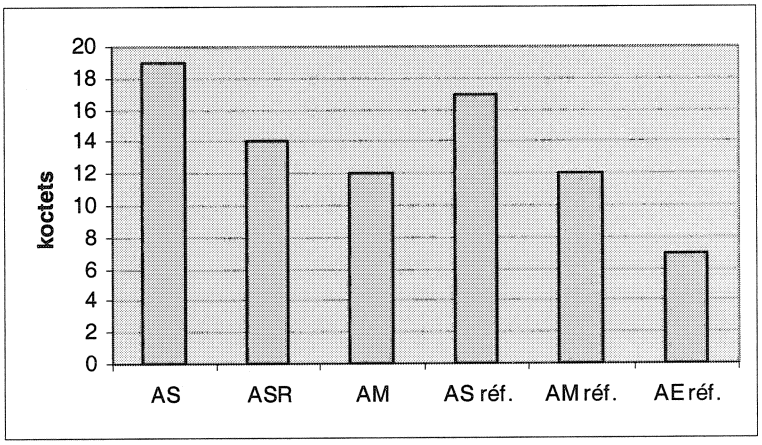

Figure 4.16 Comparaison des tailles des agents

CHAPITRE V

CONCLUSION

Ce mémoire apporte une contribution au paradigme des agents mobiles. En particulier, il s'est fixé pour objectif d'améliorer une des approches proposées dans la littérature en vue de la protection des agents mobiles contre les plates-formes malicieuses. Dans ce chapitre, nous allons faire la synthèse des travaux complétés dans le cadre de ce mémoire. Ensuite, nous présenterons les limitations de nos propositions et finalement nous exposerons nos recommandations en vue des travaux futurs.

5.1 Synthèse des travaux et améliorations apportées

Nous avons spécifié et implémenté un protocole qui se veut une contribution dans le domaine de la sécurité des agents mobiles. Nous avons conçu notre protocole pour combler les insuffisances et limitations que présente un protocole (de base) [ELR02]. Notre protocole combine en même temps les techniques de comportement de référence, de coopération entre agents, de cryptographie, d'exécution limitée dans le temps. Aussi, se veut-il un protocole réactif qui s'adapte à l'environnement et aux circonstances dans lesquels il est opérationnel.

Le protocole, ainsi conçu et implémenté, utilise le comportement de référence qui permet d'exécuter l'agent mobile en partie (code critique) ou en totalité dans un environnement qui ne présente pas d'hostilité (plates-formes de confiance). Le principe de coopération associée au premier aspect rend cette vérification de l'exécution plus dynamique que jamais. Et de là, la détection de plusieurs attaques est devenue instantanée. Le chiffrement et la signature des communications entre les agents rendent la coopération plus sure et confidentielle, ce qui augmente d'un cran la sécurité de l'agent mobile. Aussi, l'exécution limitée dans le temps ne peut qu'éviter toute exécution illégitime de l'agent mobile, ce qui a rendu possible la détection d'attaque. La

réactivité du protocole face à son environnement, en s'adaptant dynamiquement aux caractéristiques des plates-formes qui exécutent l'agent mobile et au réseau de communication, lui offre une originalité quant à l'estimation de la valeur du *Timeout* en vue de la détection de la ré-exécution de code. Cette estimation prend en charge des paramètres de la plate-forme (performance, charge,...) sans pour autant interroger une plate-forme malicieuse et risquer d'avoir des informations qui peuvent être trompeuses. Ainsi, l'intégrité et la confidentialité de l'agent mobile ne sont pas compromises. De plus, nous avons pu voir, au chapitre 4, la qualité de l'estimation dans les deux scénarios de test considérés.

Par ailleurs, nous avons doté le protocole d'une mesure de détection de l'attaque de modification permanente de code. Cette méthode est réactive tout comme celle du déni de service, car elle permet de pallier et de redresser les dégâts causés par la modification permanente de code. Ainsi, elle permet de protéger le protocole contre une telle attaque.

À quoi va servir un protocole de sécurité qui est basé principalement sur la coopération des agents, si cette coopération n'est plus possible ? C'est la question que nous nous sommes posé au départ. Ainsi, nous nous sommes rendu compte qu'une éventuelle panne de la plate-forme de confiance empêchera certainement la coopération, entraînant ainsi le blocage du protocole. La mesure que nous avons mise en œuvre pour pallier le problème a consisté à ajouter un deuxième agent qui va guetter la panne du premier et immédiatement après prendre la relève. Ceci garantira une continuité du protocole et assurera la relève. Ainsi, l'agent continuera sa mission en toute sécurité.

Par ailleurs, les spécifications étant faites, nous avons procédé à la vérification formelle de notre protocole, étape qui a bien montré sa nécessité dans la validation des logiciels, protocoles et systèmes. De nombreux exemples de systèmes classiquement testés et non vérifiés formellement ont présenté malheureusement des échecs [BÉA99]. Nous avons adopté la méthode du model-checking (vérification de modèle) pour modéliser notre protocole par le biais de ses acteurs principaux. Ainsi, nous avons construit un réseau d'automates temporisés et synchronisés qui nous a permis de représenter les différentes fonctionnalités et mécanismes de notre protocole. Nous avons

utilisé le model-checker UPPAAL qui nous a permis de représenter aussi l'aspect temporel pertinent dans notre recherche.

Avec la logique CTL, nous avons pu spécifier les propriétés que doit satisfaire notre protocole. Nous les avons regroupées en catégorie selon qu'elles expriment survivabilité/disponibilité, vivacité ou intégrité du protocole. À l'aide de vérificateur de l'outil, nous avons pu vérifier toutes les propriétés avec succès ce qui nous assure de la qualité et de la fiabilité de la modélisation.

Une originalité de ce travail réside dans le fait que nous avons conçu un processus qui modélise une plate-forme attaquante, aspect non disponible dans le système d'agents que nous avons utilisé pour l'implémentation. Ce processus nous a permis aussi de simuler des pannes de la plate-forme de confiance. Il nous a été possible de générer aléatoirement des pannes et de vérifier le comportement du protocole qui répond bien aux spécifications et à nos attentes.

Enfin, nous avons utilisé la plate-forme d'agents mobiles Grasshopper pour implémenter notre protocole. Cela nous a permis de tester les fonctionnalités et les mécanismes du protocole. Nous avons ensuite expérimenté l'estimation du *Timeout* conçue lors de la spécification du protocole dans les deux scénarios considérés. Après quoi, nous nous sommes comparé au protocole de base dans un environnement qui attaque par une exécution de code ; les résultats étaient très satisfaisants. Puis, nous avons effectué des tests pour estimer le *TimeoutAS,* la valeur du temporisateur qui permet de détecter l'interruption de la coopération primaire (la panne de *T* et donc de l'*AS*). Ensuite, nous avons testé les mécanismes de reprise après un déni de service, une modification permanente de code et également la reprise de l'*ASR* après la panne de la plate-forme de confiance *T*. Enfin, nous avons mesuré le trafic généré pour évaluer le prix à payer en contrepartie. Nous avons pu remarquer que notre protocole augmente le trafic de 13% par rapport au trafic total généré et qui découle de l'ajout de l'*ASR*.

5.2 Limitations des travaux

Si notre protocole possède plusieurs points forts quant à ses mécanismes de détection et protection des attaques, sa robustesse et sa survivabilité, il présente aussi certaines limitations. En effet, il ne permet pas de protéger l'agent mobile contre des attaques plus ingénieuses comme celles discutées à la fin du chapitre 3. Quant à la simulation des attaques, nous avons eu recours à un changement de code de l'agent mobile pour pouvoir les générer. Ainsi, le comportement de l'agent est modifié et ne représente guère le comportement d'origine.

Nous ajoutons à cela la difficulté d'estimation de l'*IAS* (intervalle d'attente supplémentaire) pour la détection du déni de service. Aussi, il nous a été difficile d'estimer le temps de propagation entre les deux plates-formes de confiance. Car, nous voulions lui ajouter le temps d'*IAS* pour avoir la valeur du *TimeoutAS*. Ainsi, nous avons estimé le *TimeoutAS* en ne se basant que sur le temps d'inter-arrivée des messages *majASR()*. Aussi, dès qu'une panne survient sur *T*, l'*ASR* prend la relève. Cependant, il assure seul la coopération avec l'*AM* et, si jamais une panne de la plate-forme *TR* survient, le protocole se bloquera certainement.

Par ailleurs, notre protocole ajoute une exigence par rapport à la nécessité d'une deuxième plate-forme de confiance (*TR*). Cependant, nous la considérons comme un prix à payer pour avoir une disponibilité du protocole suite à une absence de la coopération primaire de l'*AS*.

Concernant le modèle construit en vue de la vérification formelle, nous n'avions pas pu mesurer son temps de réponse quand une attaque ou une panne survient. Ainsi, nous aurions pu mesurer la qualité du modèle quant à sa réactivité en circonstances hostiles ou de pannes. Aussi, nous avions remarqué une limitation au niveau du model-checker qui ne permet pas de représenter le passé exprimable par CTL. Cependant, nous avions détourné cette lacune en utilisant des variables qui mémorisent le passé. Aussi, nous notons que notre modèle permet de générer une attaque à la fois et ne combine pas entre plusieurs attaques. Néanmoins, nous signalons que le modèle avait pour objectif la

génération de attaques pris en charge par le protocole. Ainsi, nous pourrons valider le comportement du protocole en face de chacune d'elle séparément.

5.3 Indications de recherches futures

Comme indications de recherches futures, nous pouvons nous inspirer des limitations que présente notre protocole. Ainsi, afin de rendre le protocole le plus sécuritaire possible, son extension pour prendre en charge d'autres attaques sera la bienvenue. Aussi, un traitement en profondeur du temporisateur de détection du déni de service permettra d'obtenir une mesure complète de détection et de protection contre le déni de service, ce qui facilite l'estimation en partie du *TimeoutAS* qui en dépend.

Nous avons implémenté une réplication dans un seul sens entre l'*AS* et l'*ASR*. Cependant, il serait mieux de pouvoir doter le protocole d'une vraie réplication, i.e. que les deux agents *AS* et *ASR* devraient jouer des rôles identiques. Ainsi, en cas de reprise de l'*AS* après une panne de *T*, le protocole devra mettre à jour l'*AS* des événements survenus en son absence pour qu'il soit en mesure de faire la reprise suite à une éventuelle panne de *TR* (donc de l'*ASR*).

Concernant les tests d'attaques, il serait intéressant de concevoir un générateur d'attaques pour une plate-forme d'agents mobiles, à l'image de celle modélisée et vérifiée formellement dans notre protocole. Au niveau de la vérification formelle, une amélioration du modèle de la plate-forme attaquante conçu dans notre protocole serait très intéressante. À titre d'exemple, la combinaison de plusieurs attaques à la fois. Ainsi, le modèle pourrait générer d'autres attaques, ce qui permettrait de guider la conception d'un tel générateur d'attaques (pour un système d'agents mobiles). Aussi, le fait de vérifier le protocole avec un autre outil, tel CHRONOS, permettra d'exprimer toutes les propriétés exprimables par CTL; le modèle ainsi obtenu serait plus complet et plus représentatif de la réalité.

BIBLIOGRAPHIE

[ALL01] G. Allée, "Sécurité des Agents Mobiles : protocole d'enregistrement d'itinéraire par Agents Coopérants", *Mémoire de maîtrise es sciences appliquées*, École Polytechnique de Montréal, Canada, Février 2001.

[BAU99] C. Bäumer, M. Breugst, S. Choy, T. Magedanz, "Grasshopper-a universal agent platform based on OMG MASIF and FIPA standards", *First International Workshop on Mobile Agents for Telecommunication Applications* (MATA'99), Ottawa, Canada. World Scientific Publishing, USA, October 1999, pp. 1-18.

[BOR02] N.Borselius, "Mobile agent security" *in Electronics & Communication Engineering Journal*, Information Security Group, University of London, UK, October 2002, vol. 14, no. 5, pp. 211-218.

[BIE02] E. Bierman, E. Cloete, "Classification of Malicious Host Threats in Mobile Agent Computing", *in Proceeding of the 2002 annual research conference of the South Africa Institute of Computers Scientists and Information Technologists on Enablement through Technology* (SAICSIT'02), September 16-18, 2002, pp. 141-148.

[COR99] A.Corradi, R. Montanari and C. Stefanelli "Mobile Agent Protection in the Internet Environment", *Proceedings of the 23rd International Computer Software and Applications Conference*, (COMPSAC'99), IEEE Computer Society Press, Phoenix, October 1999, pp. 80-85.

[ELR02] A.Elrhazi, "Sécurité des Agents Mobiles: Protocole sécuritaire basé sur un agent sédentaire parfaitement coopérant", *Mémoire de maîtrise es sciences appliquées*, École Polytechnique de Montréal, Canada, Décembre 2002.

[FAR96] W.M. Farmer, JD Guttman, and V Swarup, "Security for Mobile Agents: Issues and Requirements", *in Proceedings of the 19th National*

Information Systems Security Conference, October 1996, pp. 591-597.

[FIP96] Foundation of Intelligent Physical Agents, 1996. Documentation en ligne: http://www.fipa.org/specifications

[FRE96] Alan O. Freier, Philip Karlton et Paul C. Kocher, "The SSL Protocol Version 3.02", *Internet Draft*, November 18, 1996. Documentation en ligne : http://wp.netscape.com/eng/ssl3/draft302.txt

[GRE98] M. S. Greenberg, J. C. Byington, T.Holding, D.G. Harper, "Mobile agents and security", *IEEE Communication Magazine*, July 1998, pp 76-85.

[GUA00] X. Guan, Y. Yang, J. You, "POM- A Mobile Agent Security Model against Malicious Hosts". *HPC-Asia'00*, Beijing, China, May 2000, pp. 1165-1166.

[HOH98] F.Hohl, "Time Limited Blackbox Security: Protection Mobile Agent From Malicious Hosts", *In Mobile Agents and security*, LNCS 1419, Springer-Verlag, 1998, pp. 92-113.

[HOH00] F.Hohl, "A Framework to Protect Mobile Agents by Using Reference States", *in Proceedings 20th International Conference on Distributed Computing Systems*, IEEE Computer Society, Los Alamitos Ed., California, 2000, pp. 410-417. Documentation en ligne: *ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2000-03/TR-2000-03.ps.gz*

[IAI02] "The IAIK Java Cryptography Extension (IAIK-JCE) 3.03", Institute for Applied Information Processing and Communication of the Technical University Graz, 2002. Documentation ligne: http://jce.iaik.tugraz.at/products/01_jce/documentation/javadoc/

[IKV98] IKV++, "Grasshopper Basics And Concepts", Berlin Germany 1998. Documentation en ligne: http://www.grasshopper.de/download/doc/GrasshopperAndStandards.pdf

[JAN99] W. A. Jansen, T. karygiannis, "Mobile Agent Security'', *NIST Special*

Publication 800-19, National Institute of Standards and Technology, August 1999.

[JAN00] W. A. Jansen, "Countermeasures for mobile agent security", *Computer Communications, Special Issue on Advanced Security Techniques for Network Protection*, Elsevier Science BV, November 2000.

[KAR00] N.M. Karnik, A.R. Tripathi, "A Security Architecture for Mobile Agents in Ajanta", *in Proceedings 20th International Conference on Distributed Computing Systems*, Los Alamitos Ed., IEEE Computer Society, California, 2000, pp. 402-409.

[MIN96] Y. Minsky, R. Van Renesse, F. Scheinder, S. Stoller, "Cryptographic Support for Fault-Tolerant Distributed Computing", *in Proceeding of the 17th ACM SIGOPS*, European Workshop, 1996, pp. 109-114.

[MIT02] N. Mitrović and U. Arronategui, "Mobile Agent Security Using Proxy-agents and Trusted Domains", *Second International Workshop on Security of Mobile Multiagent Systems* (SEMAS 2002), German AI Research Center (DFKI) Research Report, ISSN 0946-008X, July 2002. disponible à l'adresse: http://www.cps.unizar.es/~mitrovic/PUBLICATIONS/SEMAS02/SEMAS02.pdf

[NEC98] G. C. Necula, P. Lee, "Safe, Untrusted Agents Using Proof-Carrying Code", *Mobile Agents and Security*, G. Vigna Ed., LNCS 1419, Springer-Verlag, 1998, pp. 61-91.

[OMG95] OMG, "Common Facilities REP3, Request for proposal TC Document", November 10, 1997.
Documentation en ligne: http://www.omg.org/docs/orbos/97-10-05.pdf

[OMG00] OMG, "Mobile Agent Facility Specification V1.0", new edition January 2000. Documentation en ligne: http://mobilitools.objectweb.org/MobiliTools/SMI/doc/SMI/MAF.pdf

[ORD96] J. J. Ordille, "When agents roam, who can trust?", *in Proceedings of the first Conference on Emerging Technologies and Applications in communications*, Portland, Oregon, Mai 1996

[ROT98] V. Roth, "Mutual Protection of Co-Operating Agents", in *Secure Internet Programming*, Vitek et Jensen (Ed.), Springer-Verlag, Berlin, Allemagne, 1998, pp. 26-37

[RIV78] R. L. Rivest, A. Shamir et L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", *Communications of the ACM*, Vol.21, Issue 2, February 1978, pp. 120-126.

[SAM02] A. Sameh, D. Fakhry, "Security in Mobile agent System", *2002 Symposium on Applications and the Internet* (SAINT'02), Jan.28-Feb.1st 2002, Nara City, Japan, IEEE Computer Society 2002, pp. 4-5.

[SAN98] T. Sander and C.F. Tschudin, "Protecting Mobile Agents Against Malicious Hosts", *in Mobile Agents and Security,* G. Vigna Ed. LNCS 1419, Springer-Verlag, 1998, pp. 44-60.

[UPP01] G. Behrmann, A. David, K. G. Larsen, O. Möller, P. Pettersson et W. Yi, "UPPAAL - Present and Future", *in Proceedings of the 40th IEEE Conference on Decision and Control (CDC'2001),* Orlando, Florida, USA, December 4 to 7, 2001

[UPP02] A. David, G. Behrmann, K. G. Larsen et W. Yi "A Tool architecture for the next generation of UPPAAL", *in "Formal Methods at the Crossroads: from Panacea to Foundational Support"*, Bernhard K. Aichernig and Tom. Maibaum (Ed), *proceedings of UNU/IIST 10th Anniversary Colloquium,* Lisbon, Portugal. Mars 2002, volume 2757 de *"Lecture Notes in Computer Science"*. Springer-Verlag, 2003, pp 352-366

[VIG98] G. Vigna, "Cryptographic Traces for Mobile Agents", *In Mobile agents and security,* G. Vigna Ed., LNCS 1419, Springer-Verlag, 1998, pp. 137-153.

[ZHU99] P.Zhu, "Research on locating and Communication services of mobiles Object system", *Ph.D thesis*, Shanghai Jiaotong Univ., 1999.

ANNEXE A

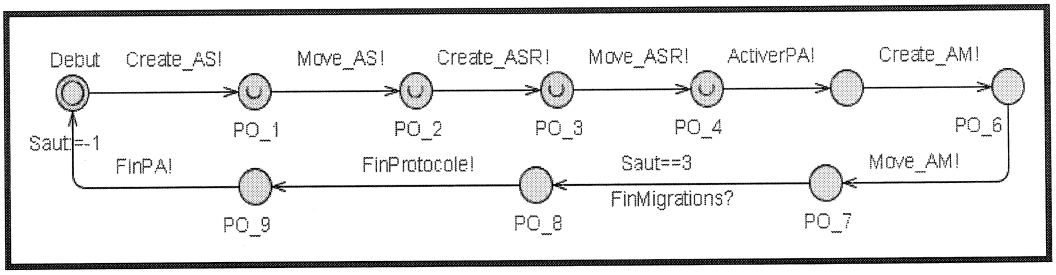

Figure A.1 Automate temporisé du processus plate-forme d'origine *PO()*

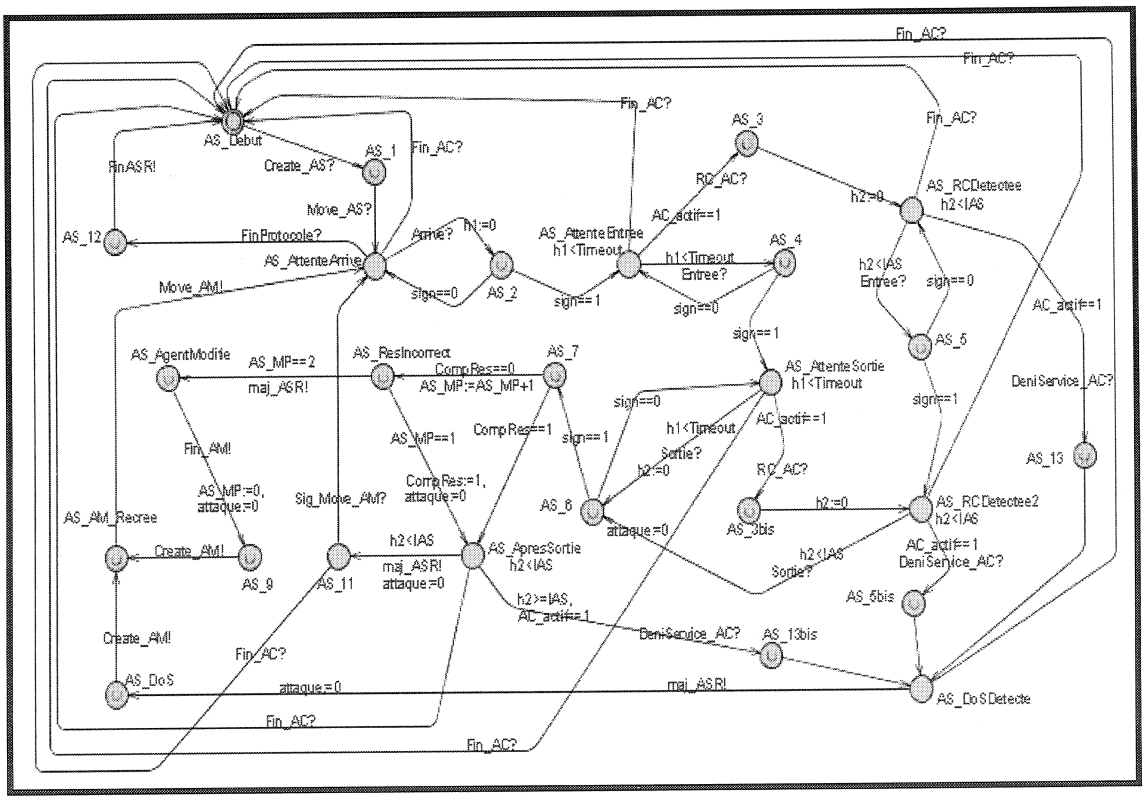

Figure A.2 Automate temporisé du processus Agent sédentaire *AS()*

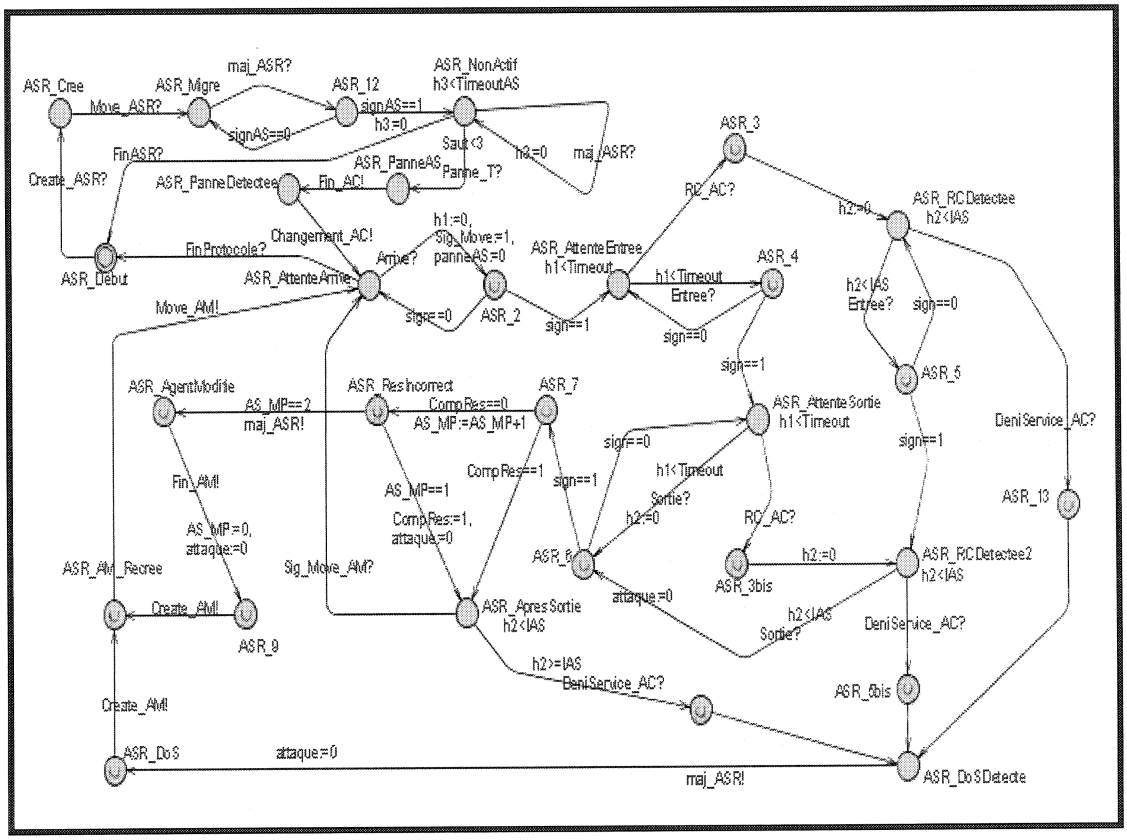

Figure A.3 Automate temporisé du processus Agent sédentaire de reprise *ASR()*

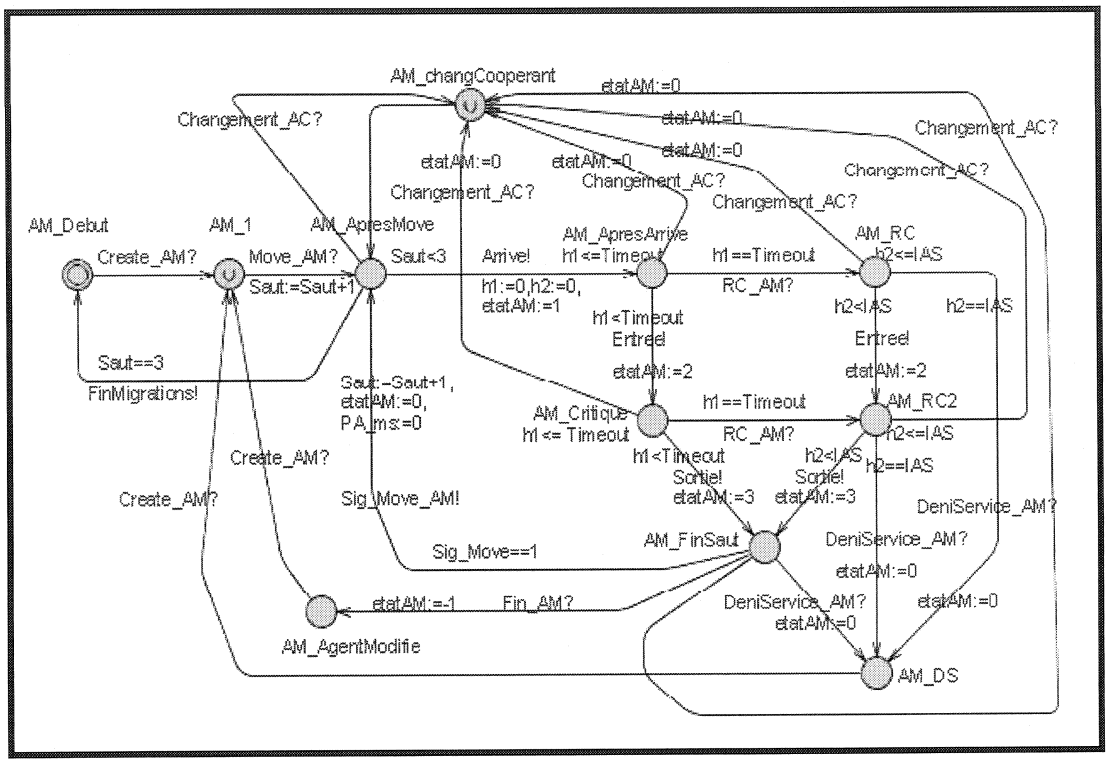

Figure A.4 Automate temporisé du processus Agent mobile *AM()*

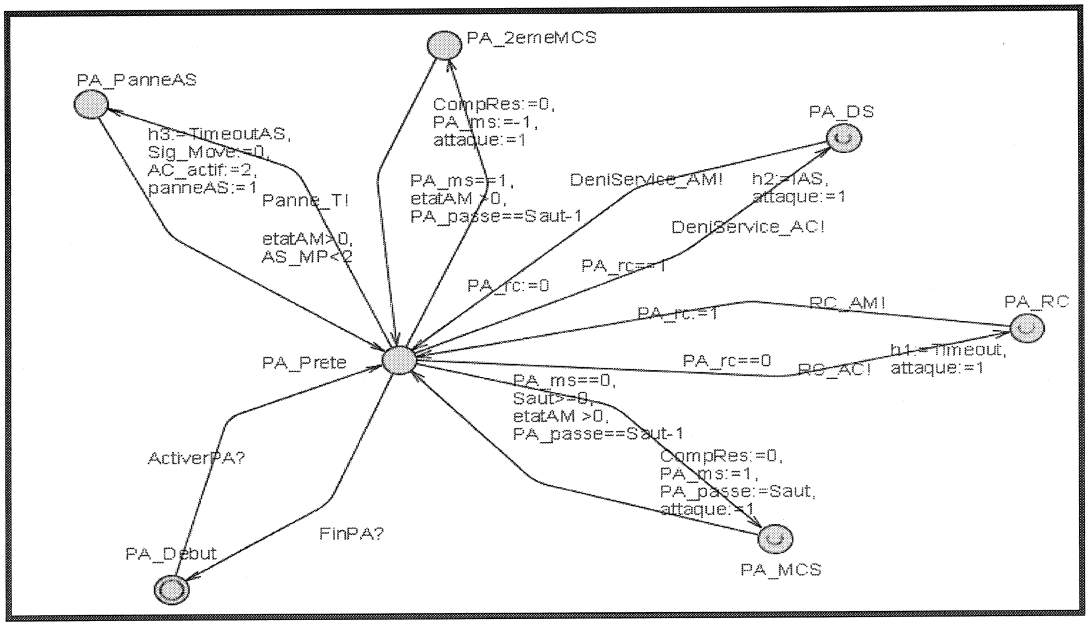

Figure A.5 Automate temporisé du processus plate-forme attaquante *PA()*